RODRIGO AMADOR COELHO

IMPLEMENTAÇÃO DE UMA REDE NEURAL ARTIFICIAL DE CAMADA SIMPLES EM PLATAFORMA GPU UTILIZANDO LINGUAGEM CUDA

LAVRAS - MG

2011

RODRIGO AMADOR COELHO

IMPLEMENTAÇÃO DE UMA REDE NEURAL ARTIFICIAL DE CAMADA SIMPLES EM PLATAFORMA GPU UTILIZANDO LINGUAGEM CUDA

Monografia apresentada ao Colegiado do Curso de Ciência da Computação, para a obtenção do título de Bacharel em Ciência da Computação.

Orientador

Prof. Dr. Wilian Soares Lacerda

LAVRAS - MG

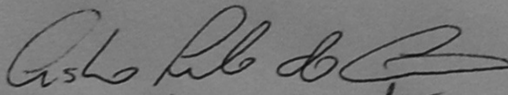
2011

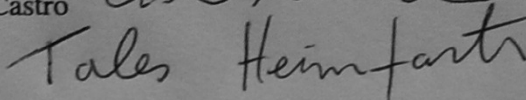
RODRIGO AMADOR COELHO

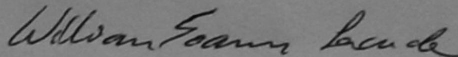
IMPLEMENTAÇÃO DE UMA REDE NEURAL ARTIFICIAL DE CAMADA SIMPLES EM PLATAFORMA GPU UTILIZANDO LINGUAGEM CUDA

Monografia apresentada ao Colegiado do Curso de Ciência da Computação, para a obtenção do título de Bacharel em Ciência da Computação.

Aprovada em *06 de Junho de 2011*

Prof. Dr. Cristiano Leite de Castro 

Prof. Dr. Tales Heimfarth 



Prof. Dr. Wilian Soares Lacerda

Orientador

LAVRAS - MG

2011

Dedico esse trabalho aos meus pais, Salustriano e Gilca e ao meu irmão, Tiago.

AGRADECIMENTOS

Agradeço aos meus pais, Salustriano e Gilca, pelo apoio que me deram.

Agradeço ao meu irmão, Tiago, pela força nos momentos mais difíceis.

Agradeço ao meu amigo da república, Ronaldo pela amizade.

Agradeço aos meus amigos do curso, pelo companheirismo e amizade durante toda a faculdade.

Agradeço aos meus amigos, que mesmo estando longe, me apoiaram, me deram forças e conselhos.

Agradeço aos professores, pelos conhecimentos repassados.

Agradeço à professora, Marluce, pela orientação neste trabalho.

RESUMO

Essa monografia descreve técnicas de paralelização para implementação de redes neurais. Estão sendo utilizadas técnicas de programação em GPU (Graphic Processing Units ou Unidade de Processamento Gráfico) para fins de melhoria de performance em redes neurais artificiais. Duas diferentes técnicas foram implementadas para serem executadas junto a GPU, mas não obtiveram ganho frente a CPU. Uma análise completa do ganho de velocidade e das técnicas utilizadas mostrando qual a mais eficiente é exibido no final deste estudo.

Palavras-chave: Rede naural Artificial, Paralelismo, GPU, CUDA, OpenCL

ABSTRACT

This monograph describes parallelization techniques for implementation of neural networks. Techniques are being used for GPU programming in order to improve performance in artificial neural networks. Two different techniques were implemented to run from the GPU, but have been unsuccessful against the CPU. A complete analysis of the speedup of the used techniques shows which is the more efficient is displayed at the end of the study.

Keywords: Artificial Neural Network, Parallelism, Artificial Inteligence, GPU, CUDA, OpenCL

LISTA DE FIGURAS

Figura 1 Neurônio Genérico (Fonte: Tatibana e Kaetsu, 2009) 14
Figura 2 Neurônio Artificial desenvolvido por McCulloch (Fonte: Tatibana e Kaetsu, 2009) 14
Figura 3 Exemplo de uma RNA com 4 camadas (Fonte: Tatibana e Kaetsu, 2009) 15
Figura 4 Perceptron (Fonte: Tatibana e Kaetsu, 2009) 17
Figura 5 Pseudo código de um Perceptron 17
Figura 6 Hierarquia de threads, bloco e memória(Fonte: NVIDIA *Corporation*) 21
Figura 7 Hierarquia de Grids e memória(Fonte: NVIDIA *Corporation*) 22
Figura 8 Comportamento de um programa CUDA (Fonte: NVIDIA *Corporation*) 23
Figura 9 kernel GPU 24
Figura 10 Inicialização das variáveis 25
Figura 11 Cópias de memória e chamada do kernel da GPU 26
Figura 12 Impressão de resultados e liberação de espaço alocado na GPU 27
Figura 13 Comparativo as terminologias para implementação(Fonte: AMD *Developer Central*) 28
Figura 14 Comparação entre comando CUDA x OpenCL (Fonte: AMD *Developer Central*) 28
Figura 15 Leitura do programa GPU-Z da GPU GTX285 33
Figura 16 Representação gráfica da rede neural implementada 34
Figura 17 Representação gráfica do dígito do número um 34
Figura 18 Algoritmo do código 36
Figura 19 Relatório gprof 37
Figura 20 Função *AdjustWeights* CPU 37
Figura 21 Sequencial GPU (1 *thread*) 38
Figura 22 Sequencial GPU (N *thread*) 39
Figura 23 Componentes de uma thread 39
Figura 24 Algoritmo do método do neurônio por *thread* 40
Figura 25 Treinamento 11 neurônios 42
Figura 26 Treinamento 22 neurônios 43
Figura 27 Treinamento 34 neurônios 44
Figura 28 Treinamento 11 neurônios 45
Figura 29 Treinamento 22 neurônios 46
Figura 30 Treinamento 34 neurônios 47
Figura 31 Tempo do Crivo 57

LISTA DE TABELAS

SUMÁRIO

1	**Introdução**	10
1.1	**Contextualização e Motivação**	10
1.2	**Objetivos**	11
1.3	**Estrutura**	11
2	**Referencial Teórico**	13
2.1	**Redes Neurais Artificiais**	13
2.2	**Perceptron**	16
2.3	**Computação Paralela**	18
2.4	**Programação em GPGPU**	19
2.5	**CUDA**	20
2.6	**ATI *Stream***	26
2.7	**Trabalhos Realizados nos Últimos Anos**	28
3	**Materiais e Métodos**	32
3.1	**Tipo de Pesquisa**	32
3.2	**Materiais**	32
3.3	**Procedimentos Metodológicos**	32
3.3.1	**Método da menor granularidade**	35
3.3.2	**Método do Neurônio por *thread***	38
4	**Resultados**	41
4.1	**Resultados Obtidos com o Método da Menor Granularidade**	41
4.2	**Resultados Obtidos com o Método do Neurônio por *Thread***	45
5	**Conclusões**	49
5.1	**Propostas de Continuidade**	49
6	**Referencia Bibliográfica**	51
7	**ANEXO Crivo de Erastóstenes**	56
8	**ANEXO Código Crivo de Eratóstenes**	58
9	**ANEXO Códigos de RNA**	64

1 Introdução

1.1 Contextualização e Motivação

O homem em sua busca pela perfeição e velocidade sempre se inspirou na natureza para solucionar problemas. Não foi diferente no mundo da computação, onde inspirados no cérebro humano, os pesquisadores McCulloch e Pitts (1943), Hebb (1949) e Rosenblatt (1961) tiveram as mais importantes publicações que introduziram o primeiro modelo de redes neurais artificiais.

Redes neurais são utilizadas em uma grande variedade de aplicações, tais como reconhecimento de voz (Scanzio *et al.*, 2010), reconhecimento de padrões (Jang *et al.*, 2008), reconhecimento de face (Poli *et al.*, 2008), avaliação de crédito (Khashman, 2010), robótica (Shi *et al.*, 2010), entre outras.

Redes neurais podem ser implementadas em hardware (Misra and Saha, 2010). Um exemplo de implementação em hardware é o mapeamento da rede neural em uma arquitetura de *hardware* de alto desempenho, como uma plataforma *Field-Programmable Gate Array* (FPGA). Ly e Chown (2010) propuseram um *framework* modular de uma máquina de Boltzmann restrita (RBM) buscando reduzir a complexidade de tempo das computações. RBMs grandes foram particionadas em componentes menores e distribuídos entre múltiplo recursos de FPGA e assim conseguiram obter desempenho computacional.

A execução de grande quantidade de dados utilizando uma rede neural poderá ter um tempo de execução alto, o que motiva a utilização de técnicas que acelerem o processamento de uma rede neural.

Como o nosso cérebro não funciona de modo sequencial, evoluiu-se o conceito de programação, onde as tarefas computacionais são executadas de forma sequencial ou linear, muitas vezes deixando o hardware ocioso, surgindo o conceito

de paralelismo. Na computação paralela, tarefas que não possuem dependência entre si podem ser executadas de forma concorrente, aproveitando melhor os ciclos de CPU (*Central Processing Unit*) e acelerando a execução da aplicação.

A tecnologia do *hardware* também tem evoluído bastante nos últimos anos, com arquiteturas altamente paralelas e com alto poder de processamento a um custo financeiro relativamente baixo. Empresas como a IBM com o processador CELL (Gschwind, 2005), a NVIDIA com as placas de vídeo com múltiplas unidades de processamento (CUDA, 2009) e a ATI com a tecnologia *Stream Processor* (ATI, 2009) têm mostrado um novo caminho para os programadores que enfrentavam dificuldades em implementar soluções de alto desempenho.

Com esta nova tecnologia de hardware surgiu também a motivação para explorar o paralelismo de redes neurais artificiais, visando reduzir o tempo de execução e a obtenção mais rápida de resultados.

1.2 Objetivos

Este trabalho tem como objetivo geral o estudo de diferentes técnicas de programação paralela para se implementar redes neurais artificiais em GPU (Graphic Processing Units) utilizando CUDA(*Compute Unified Device Architecture*).

Como objetivo específico serão propostas diferentes implementações paralelas para rede neural artificial para serem executadas em GPU utilizando CUDA, visando obter alto desempenho.

1.3 Estrutura

Os capítulos seguintes desta monografia estão organizados da seguinte forma: O capítulo 2 aborda os principais conceitos referentes às redes neurais artificiais, programação paralela, programação em placas de vídeo e trabalhos rea-

lizados nesta área . O capítulo 3 apresenta toda a metodologia de trabalho. Já o capítulo 4 apresenta os resultados e os discute. O capítulo 5 conclui o trabalho e sugere trabalhos futuros. Os capítulos de anexo trazem os códigos utilizados para a realização de todo este trabalho e um algoritmo com ganho de desempenho da GPU frente a CPU com o Crivo de Erastóstenes.

2 Referencial Teórico

Este capítulo aborda os conceitos relacionados a rede neural artificial, o perceptron, computação paralela, programação para GPGPU, CUDA, ATI Stream e os trabalhos relacionados.

2.1 Redes Neurais Artificiais

Uma Rede Neural Artificial (RNA) é um modelo baseado na natureza, mais especificamente no cérebro humano (Barreto, 2002). O cérebro humano é composto por cerca de 10 bilhões neurônios. Os neurônios se conectam através de sinapses, e juntos formam uma grande rede chamada de rede neural (Tatibana e Kaetsu, 2009). A Figura 1 apresenta uma ilustração de um neurônio biológico genérico e suas partes constituintes.

Uma rede neural artificial (RNA) é um sistema composto por vários neurônios. Alguns neurônios recebem excitações do exterior e são chamados neurônios de entrada. Outros têm suas respostas usadas para alterar, de alguma forma, o mundo exterior e são chamados neurônios de saída. Os neurônios que não são nem entrada nem saída são conhecidos como neurônios internos. Estes neurônios internos à rede têm grande importância e são conhecidos na literatura saxônica como *"hidden"*, traduzindo para o português, "escondido" (Barreto, 2002). A figura 2 apresenta um neurônio artificial desenvolvido por McCulloch em 1943. Neste neurônio, as entradas são combinadas utilizando uma função F visando produzir um estado de ativação do neurônio.

A Figura 3 mostra uma RNA com 2 neurônios na camada de entrada, 8 neurônios na camada escondida e 1 neurônio na camada de saída.

As Redes Neurais Artificiais ao longo do tempo foram evoluindo tanto em complexidade quanto em poder de aprendizado. Primeiramente surgiu o Per-

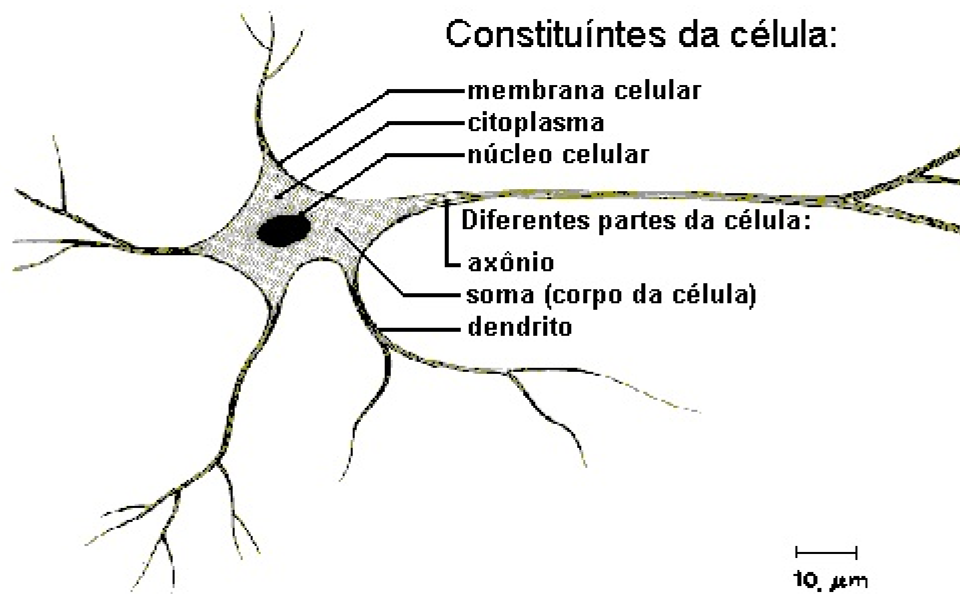


Figura 1: Neurônio Genérico (Fonte: Tatibana e Kaetsu, 2009)

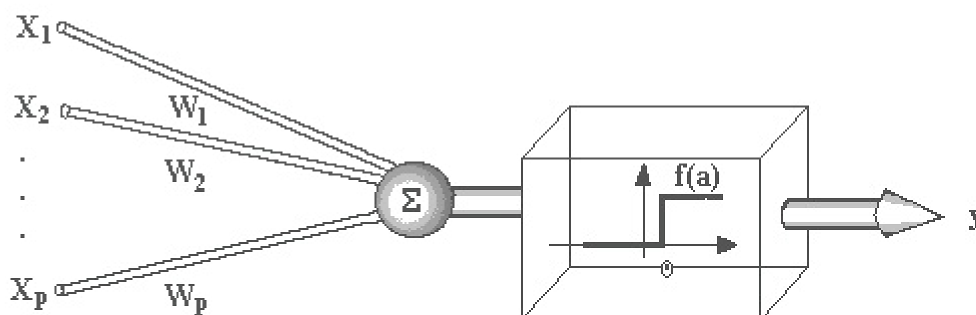


Figura 2: Neurônio Artificial desenvolvido por McCulloch (Fonte: Tatibana e Kaetsu, 2009)

ceptron de Rosenblatt (1961), onde os neurônios eram organizados em camada de entrada e saída, onde os pesos das conexões eram adaptados a fim de se atingir a eficiência sináptica.

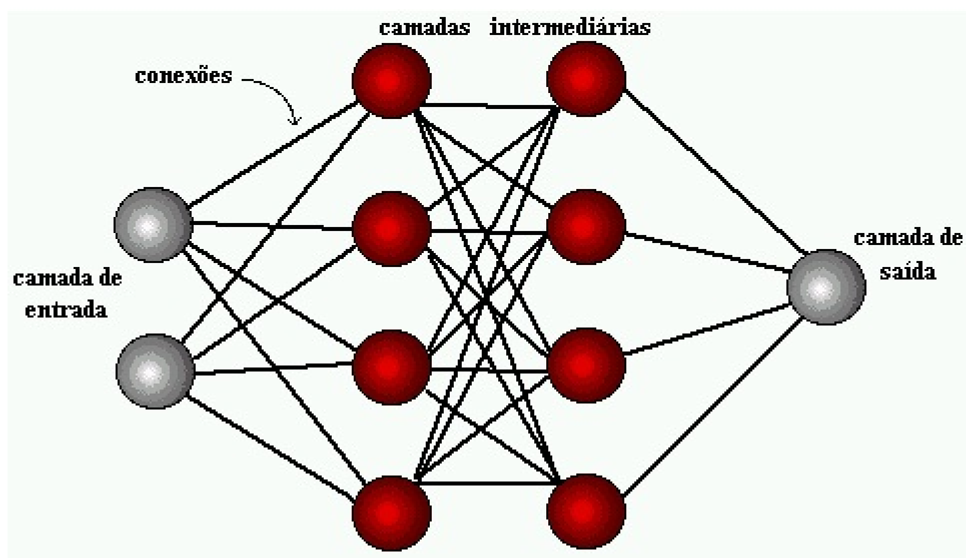


Figura 3: Exemplo de uma RNA com 4 camadas (Fonte: Tatibana e Kaetsu, 2009)

Surgiu a rede ADALINE (ADAptative LInear Network) e posteriormente o MADALINE(Many ADALINE) perceptron, proposto por Widrow e Hoff. Estas redes neurais artificiais utilizavam-se de saídas analógicas em uma arquitetura de três camadas. Rumelhart, Hinton e Williams introduziram o método Backpropagation. Era o modelo que apresentava a camada oculta, que poderiam ser várias, e um algoritmo onde a retro-propagação do erro sobre as camada ocultas para o reajustes dos pesos sugeriu seu nome.

As redes neurais artificiais podem ser aplicadas em diversos problemas. Atualmente, já vêm sendo aplicadas em (Tatibana e Kaetsu, 2009):

- análise e processamento de sinais;
- controle de processos;
- robótica;

- classificação de dados;
- reconhecimento de padrões em linhas de montagem;
- filtros contra ruídos eletrônicos;
- análise de imagens;
- análise de voz;
- avaliação de crédito.

2.2 Perceptron

Criado por Rosenblatt(1961), o perceptron é um clássico modelo de neurônio que permite saídas diretas muito usado para tarefas de classificação. O processo de classificação se baseia em encontrar padrões em comum que possam ser linearmente separadas em classes. A figura 4 mostra o modelo de um Perceptron com várias entradas, para cada entrada um peso. É usada uma função somatória com um valor de entrada (*threshold*) e uma função de ativação, no caso da figura, sigmoidal.

A regra Delta usada para o treinamento do Perceptron ajusta os pesos somente quando um padrão é classificado incorretamente. Na figura 5 perceber que dentro da estrutura de repetição na linha 3 é feito o calculo para a possível saida. Na estrutura da linha 4, é feita a regra delta para se obter o valor da saída. Na linha 8 é verificada se a saída calculada está correta e o erro é calculado. Na linha 9 é feita a atualização do peso.

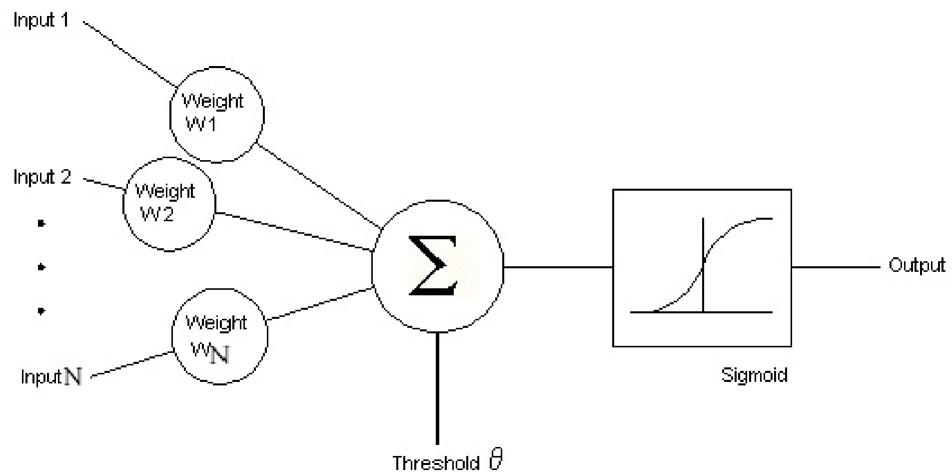


Figura 4: Perceptron (Fonte: Tatibana e Kaetsu, 2009)

```
perceptron (){
1  While (gerção != MAX){
2    for (j=0; j<=Quantidade; j++) {  //para cada padão de entrada
3      U = Peso * Entrada;            //calculo a saida
4      if (U >= 0)                    // regra Delta
5        Saida[i] = 1;
6      else
7        Saida[i] = 0;
8      ERRO = SaidaDesejada - Saida;   //verifico o erro
9      Peso = Alfa * ERRO * ENTRADA;   //atualizo o Peso
10    }
11  }
12}
```

Figura 5: Pseudo código de um Perceptron

2.3 Computação Paralela

A transição entre a computação sequencial e a paralela é uma forma encontrada para utilizar de forma mais consciente os recursos computacionais disponíveis. Assim, a computação paralela vem minimizar o tempo de computação, racionalizando melhor o uso do hardware e economizando recursos (Marowka, 2008).

Computador paralelo é um computador ou vários deles equipados com processadores capazes de trabalhar de forma cooperativa para resolver um problema computacional. Como exemplo temos os super computadores com centenas ou milhares de processadores, redes de estação de trabalho, múltiplas estações de trabalho e sistemas incorporados. Computadores paralelos são interessantes porque oferecem uma possibilidade de compartilhar os recursos computacionais, sejam processadores, memória, entrada e saída de dados, largura de banda para resolver inúmeros problemas computacionais (Foster, 1995).

Dividir o problema em problemas menores, distribuir esses problemas menores entre os processadores ou computadores, implementar um mecanismo de comunicações e sincronia entre os mesmos são apenas alguns dos passos que devem ser seguidos para que um problema seja paralelizado.

As redes neurais são paralelas por natureza, logo as redes neurais artificiais podem ter seu algoritmo paralelizado. O quanto o algoritmo pode ser paralelizado irá determinar a sua granularidade. Em granularidades mais finas, cálculos básicos do algoritmo são executados em paralelos, já a granularidade mais grossa toda uma estrutura do algoritmo é executada em paralelo.

2.4 Programação em GPGPU

A Unidade de Processamento Gráfico ou GPU (*Graphic Processing Unit*) é um processador especializado em processamento gráfico. Ele é facilmente encontrado em videogames, computadores pessoais, estações de trabalho, em celulares e em tudo que tenha algum *display* que exiba imagens mais elaboradas.

Com o interesse por exibição gráfica crescendo, com a alta resolução e o alto desempenho, começaram a ser criadas GPUs com processamento 2D em *chip* único, anos depois GPUs com processamento 3D. Com isso as GPUs receberam a capacidade de processar mais cálculos de ponto flutuante, capacidade de processar figuras geométricas, tornando-se tão flexíveis quanto uma CPU (Mocelin *et al.*, 2009).

Logo que surgiram as primeiras ferramentas para se utilizar a GPU além de sua função básica, surgiu o conceito de GPGPU (*General-Purpose computation on Graphics Processing Units* ou Unidade de processador Gráfico de Propósito Geral) com suporte para interfaces de programação e linguagens padrão da indústria, tais como C. Assim as aplicações para GPU possam ser executadas em menor tempo do que as aplicações para CPU usando o que as GPUs tem de melhor, o paralelismo (GPGPU.org, 2009).

A NVIDIA desenvolveu uma arquitetura de computação paralela chamada CUDA (*Compute Unified Device Architecture*), que tira proveito do mecanismo de computação paralela das GPUs. Também foi disponibilizado todo um kit de desenvolvimento baseado em linguagem de alto nível C com um compilador proprietário para o desenvolvimento de softwares de alta performance compatíveis com CUDA. Com o nome de ATI *Stream* SDK, a sua concorrente ATI também disponibiliza tecnologia semelhante.

2.5 CUDA

Em Novembro de 2006 a NVIDIA introduziu uma arquitetura de computação paralela de propósito geral, com um novo modelo de programação paralela que aproveita o mecanismo de computação paralela em GPUs para resolver muitos problemas computacionais complexos em uma maneira mais eficiente do que em um CPU. O CUDA (*Computer Unified Device Architecture*) esconde a complexidade da GPU através de sua API (*Application Programming Interface* ou Interface de Programação de Aplicações), permitindo assim que os programadores não se preocupem com detalhes complexos de *hardware* durante a programação.

O modelo de programação paralela CUDA é projetado para programadores familiarizados com o padrão da linguagem C. Em sua essência, as abstrações adicionadas são uma hierarquia de *threads*, memórias compartilhadas e uma sincronização para as *threads*.

A linguagem C para o CUDA foi estendida permitindo que o programador defina funções chamadas de *kernel*.A palavra *kernel* é utilizada para determinar que certo trecho do código será utilizada pela GPU. O código que será executado na GPU é separado do restante pois possui prefixos especiais e chamadas reservadas, identificando o que será processado pela CPU e pela GPU.

O modelo de programação para a GPU segue uma hierarquia de *threads* e de memória. Cada *thread* possui uma memória local, e um conjunto de *threads* é organizado em blocos, e cada bloco de *thread* possui uma memória compartilhada para todas as *threads*, como é mostrado figura 6.

Cada bloco de *threads* por sua vez agrupa se em *Grid* e as *Grids* compartilham de uma mesmo espaço de memória, chamado memória global como ilustra a figura 7.

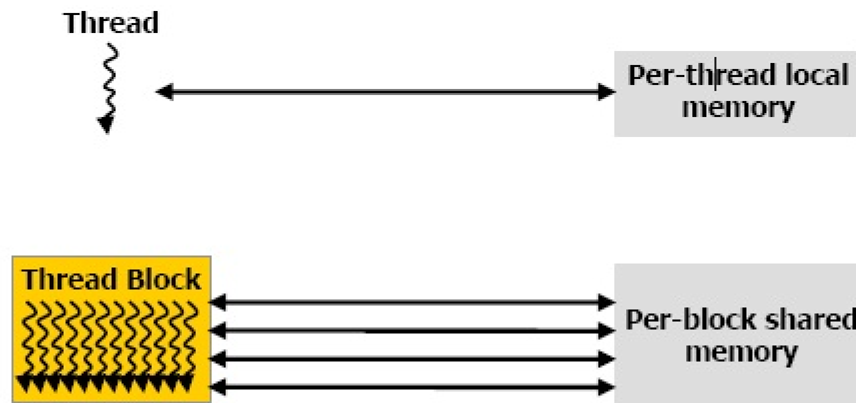


Figura 6: Hierarquia de threads, bloco e memória(Fonte: NVIDIA *Corporation*)

Toda essa hierarquia de *threads*, blocos e *grids* com suas respectivas memórias devem ser respeitadas para cada *kernel* executado.

Uma visão geral de como se comporta um programa CUDA é demonstrado na figura 8. O programa seqüencial é inicializado pela CPU, o CUDA reconhece a CPU como Host. Durante a execução desse programa, existem chamadas para um *kernel* paralelo, este *kernel* ativa um *Device*, que é como a GPU é reconhecida. Quando ativada, a GPU passa a executar um trecho do programa de forma paralela, enquanto isso a CPU continua seu trabalho de forma seqüencial. Ao fim do processo paralelo a GPU retorna com o resultado de seu processamento para a CPU.

No programa exemplo da figura 9 estão presentes os principais comandos e estruturas de um programa CUDA. Pode-se observar que a primeira palavra reservada

```
__global__
```

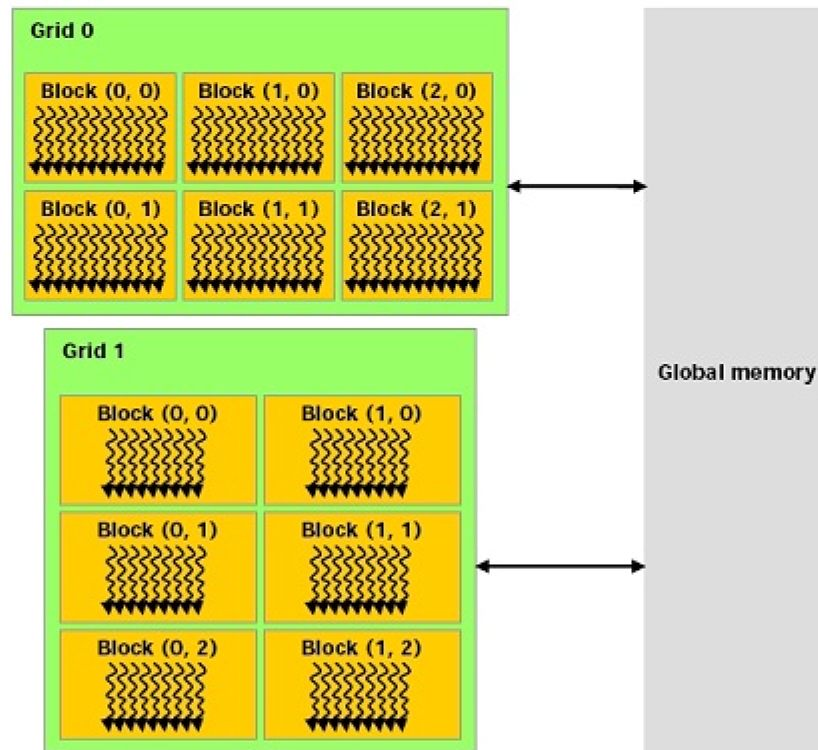


Figura 7: Hierarquia de Grids e memória(Fonte: NVIDIA *Corporation*)

é usada para determinar o que será executado pela GPU, a palavra reservada antecipa a função. Estão presentes três palavras reservadas para controle e identificação das *threads*

`blockIdx.x`

`blockDim.x`

e

`threadIdx.x`

que são respectivamente identificação do bloco, dimensão do bloco e por último a identificação da *thread*.

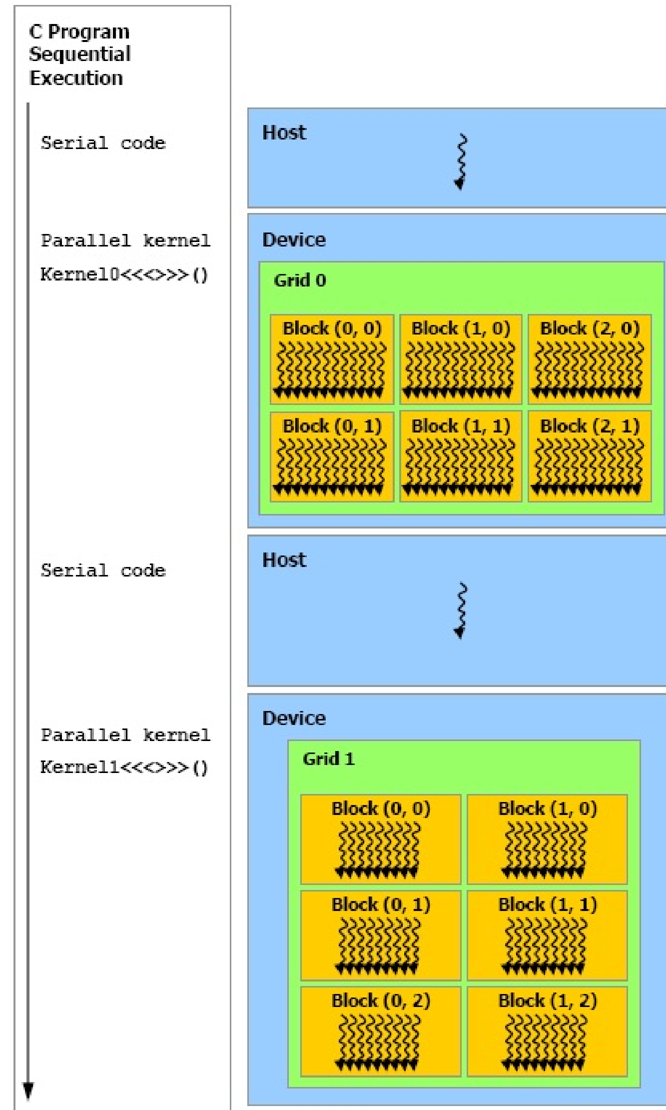


Figura 8: Comportamento de um programa CUDA (Fonte: NVIDIA *Corporation*)

A parte do código que executará seqüencialmente na CPU apresenta apenas os seguintes passos: alocação de memória, cópia das variáveis da CPU para

```
//Kernel GPU

__global__ void eleva_quadrado( int *A )

{
   int idx = blockIdx.x*blockDim.x+threadIdx.x;

   A[idx] = A[idx] * A[idx];
}
```

Figura 9: kernel GPU

GPU, chamada do *kernel* da GPU e por último a cópia da variável da GPU para CPU.

Na figura 10 podemos observar a declaração das variáveis do tipo ponteiro, determinação de seu tamanho estaticamente e um comando reservado CUDA,

`cudaMalloc()`

, que é a versão do comando de alocação de memória

`malloc()`

(em linguagem C) mas para alocação do espaço em memória na GPU.

Na figura 11, pode-se observar os três principais passos para a programação CUDA, a cópia da variável para a GPU, a chamada do *kernel* da GPU para execução paralela do código e por último a cópia da variável para a CPU. A cópia é feita através do comando reservado

```
main () // CPU
{

   int *vetorA ;
   int *d_vetorA ;
   int *d_retornoA ;
   int i;

   size_t size = sizeof(int)*256;

   vetorA= (int*) malloc(size);
   d_retornoA= (int*) malloc(size); ;

   cudaMalloc((void**)&d_vetorA, size);

   for (i=0; i<256; i++){ //preencho os vetores
      vetorA[i] = i;
   }
```

Figura 10: Inicialização das variáveis

`cudaMemcpy()`

seguido da seguinte estrutura: variável de destino, variável de origem, tamanho e sentido da cópia. O sentido da cópia é dado por outra palavra reservada

`cudaMemcpyHostToDevice`

ou

`cudaMemcpyDeviceToHost`

que copia as informações respectivamente no sentido da CPU para GPU ou GPU para CPU.

A chamada para o *kernel* na GPU é dado como uma função em C, mas com a adição de um parâmetro

`<<< , >>>`

que determina o número de blocos e *threads* ou *grids* e blocos que serão executados nesta chamada. No exemplo da figura 11 são executados 1 bloco com 256 *threads*.

```
cudaMemcpy(d_vetorA, vetorA, size, cudaMemcpyHostToDevice);

eleva_quadrado<<<1, 256>>>(d_vetorA);

cudaMemcpy(d_retornoA, d_vetorA, size, cudaMemcpyDeviceToHost);
```

Figura 11: Cópias de memória e chamada do kernel da GPU

Na figura 12 pode-se observar o comando de impressão das variáveis em tela, com seus respectivos valores e por último um comando reservado

`cudafree()`

usado para liberar o espaço estaticamente alocado para a variável na GPU.

2.6 ATI *Stream*

A tecnologia ATI *Stream* é uma tecnologia que permite que o programador tenha acesso aos núcleos da GPU possibilitando que trabalhe em conjunto

```
for (i=0; i<256; i++) {
 printf("vetorA[%d] ||| d_retornoA[%d] ||| k = %d\n", vetorA[i],d_retornoA[i], i);
   }
cudafree(d_vetorA);
}
```

Figura 12: Impressão de resultados e liberação de espaço alocado na GPU

com a CPU, para acelerar as aplicações. Assim como a NVIDIA desenvolveu o CUDA, a ATI disponibilizou uma plataforma de desenvolvimento completa, a *Stream Software Development Kit* (SDK), criada pela AMD para permitir o desenvolvimento de aplicações para a tecnologia ATI Stream. O SDK permite que se desenvolva aplicativos em uma linguagem de alto nível utilizando o OpenCL (*Open Computing Language*).

OpenCL foi inicialmente desenvolvida pela Apple, que tem direitos sobre a marca, mas hoje tem o padrão administrado pelo Khronos Group, que também gerencia OpenGL e tem como contribuintes de desenvolvimento a AMD, Apple, Intel e NVIDIA.

O funcionamento tanto do gerenciamento de *threads* quanto de memória para o OpenCL é semelhante ao CUDA. Na figura 13 há uma terminologia diferente para alguns termos na implementação com CUDA e com OpenCL, mas com a mesma funcionalidade.

Esta semelhança possibilita que os códigos em CUDA sejam portados com bastante facilidade para o OpenCL. A figura 14 apresenta os principais comandos na plataforma CUDA e sua respectiva terminologia no OpenCL.

C for CUDA terminology	OpenCL terminology
Thread	Work-item
Thread block	Work-group
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory

Figura 13: Comparativo as terminologias para implementação(Fonte: AMD *Developer Central*)

C for CUDA terminology	OpenCL terminology
__global__ function (callable from host, not callable from device)	__kernel function (callable from device, including CPU device)
__device__ function (not callable from host)	No annotation necessary
__constant__ variable declaration	__constant variable declaration
__device__ variable declaration	__global variable declaration
__shared__ variable declaration	__local variable declaration

Figura 14: Comparação entre comando CUDA x OpenCL (Fonte: AMD *Developer Central*)

2.7 Trabalhos Realizados nos Últimos Anos

Esta seção apresenta trabalhos relacionados à utilização de programação para GPUs e implementação de redes neurais artificiais paralelas, especialmente aquelas utilizando CUDA. Nestes trabalhos foram utilizadas diferentes redes neurais artificiais: Spyking, Neocognitron, celular e perceptron multi-camadas.

Fang *et al* (2009) desenvolveram um programa para mineração de dados paralela utilizando processadores gráficos. GPUminer é um novo sistema de mineração de dados para utilizar da nova geração de GPUs. O GPUminer foi projetado sobre as seguintes componentes: uma CPU para lidar com o *buffer* de entrada e saída e gerenciar a transferência de dados entre CPU e GPU; Um módulo de co-processamento paralelo entre CPU-GPU. Com esta estrutura montada conseguiram significantes *speedups* sobre a execução em uma CPU.

Pamplona *et al* (2010) apresentaram um trabalho de análise de desempenho entre a linguagem de programação C e o CUDA para implementações do problema das N-Rainhas. O problema N-Rainhas é definido quando em um tabuleiro de xadrez NxN, são posicionadas as N rainhas de maneira que não se ataquem. Dentro deste panorama foram implementados os códigos para CPU e GPU e comparados os tempos de execução de soluções para cada uma destas arquiteturas. Mesmo com maior número de processadores e da exclusividade da GPU tiveram como resultado a CPU com desempenho superior.

Jang *et al* (2008) apresentou uma série de algoritmos de redes neurais artificiais usando OpenMP e CUDA para o processamento de imagem em GPU para busca de computação mais rápida. Implementaram redes neurais artificiais baseadas em detecção de texto usando a proposta de cada arquitetura, OpenMP e CUDA. O código OpenMP demonstrou ser 10 vezes mais rápido que o código sequencial em CPU e 5 vezes mais rápido do que o código para GPU.

Nageswaran *et al* (2008) apresentaram uma eficiente simulação de uma rede neural utilizando o simulador Spyking Neural Network e uma GPU. Demonstraram um eficiente neurônio baseado numa rede neural Spyking de grande escala. Este tipo de rede neural é geralmente simulado em grandes clusters, supercomputadores ou hardwares de arquitetura dedicada. Os resultados obtidos foram 26 vezes

mais rápidos do que os resultados obtidos na versão sequencial para CPU quando simulada uma rede com 100 mil neurônios e 50 milhões de conexões sinápticas. Para um modelo de 10 milhões de conexões sinápticas foi somente 1,5 vezes mais lenta do que seria na estrutura biológica.

Poli *et al* (2008) apresentaram soluções de alto desempenho para o reconhecimento de face com uma implementação de rede neural artificial Neocognitron junto a arquitetura CUDA. Neocognitron é uma rede neural artificial proposta por Fukushima, constituída de várias camadas hierárquicas de neurônios organizadas em matrizes bidimensionais chamadas planos celulares. Utilizando da arquitetura CUDA, o balanceamento de carga foi organizado através do uso de conexões celulares com as threads se organizando em blocos, seguindo a filosofia do desenvolvimento da arquitetura CUDA. Os resultados mostraram a flexibilidade desse tipo de dispositivo como uma ferramenta de processamento paralelo em massa, quanto menor a granularidade e a dependência dos dados de processamento paralelo, melhor foi o desempenho. Conclu-se que a implementação para a arquitetura CUDA foi mais eficiente que a implementação sequencial para a CPU.

Fernandez *et al* (2008) simularam uma rede neural celular (CNN) em uma GPU com a utilização do CUDA. Os resultados mostraram uma grande eficiência devido ao alto grau de paralelismo proporcionado pela GPU. Para isso foi simulada uma rede neural celular padrão desenvolvida por Chuan-Yang, que foi otimizada para a CPU. Foi também implementada uma versão paralela para a GPU. Utilizaram esta rede neural para fazer detecção de bordas binárias, resolver a equação diferencial parcial de Laplace e propagação de onda. Os resultados demonstraram que o programa desenvolvido para utilizar o paralelismo da GPU foi mais rápido que na CPU.

Scanzio *et al* (2010) realizaram a implementação paralela de uma rede neural artificial para reconhecimento de voz para uma arquitetura de GPU. Foi utilizado o algoritmo backpropagation para padrões sequenciais e a linguagem de programação C com CUDA. Além da GPU foram realizados experimentos em CPU com múltiplas threads em múltiplos núcleos. Na comparação entre os resultados obtidos com a implementação múltiplas threads em múltiplos núcleos e a implementação para GPU demonstrou que a GPU teve melhores resultados.

3 Materiais e Métodos

Este capítulo apresenta o tipo de pesquisa utilizado neste trabalho, os métodos de implementação utilizados e a descrição de como serão realizados os experimentos e da entrada para a rede neural implementada.

3.1 Tipo de Pesquisa

Trata-se de uma pesquisa exploratória, pois visa o aprimoramento de idéias já existentes e a descoberta de novas informações. Quanto aos procedimentos pode ser classificada como pesquisa experimental, pois requer a manipulação de variáveis para a coleta de dados sobre o fenômeno de interesse, visando obter os melhores resultados.

3.2 Materiais

O equipamento utilizado para realizar os experimentos foi um computador equipado com processador Intel Core 2 Duo E8400, 3.0 GHz, FSB 1333Mhz, cache L2 6MB, soquete 775 com 4GB de memória DDR2 667Mhz e uma placa gráfica GeForce GTX 285, PCI-e 16x, barramento 512-bits, 1024MB GDDR3, core 670MHz, memória 2500MHz (Figura 15). O sistema operacional utilizado foi o Ubuntu 9.04.

3.3 Procedimentos Metodológicos

Nesta seção são apresentados os métodos utilizados para a paralelização da rede neural artificial. Para a primeira técnica de paralelização, método da menor granularidade, a rede neural artificial usada para análise de performance é uma versão modificada de uma RNA desenvolvida por Karsten Kutza(1996), pois ape-

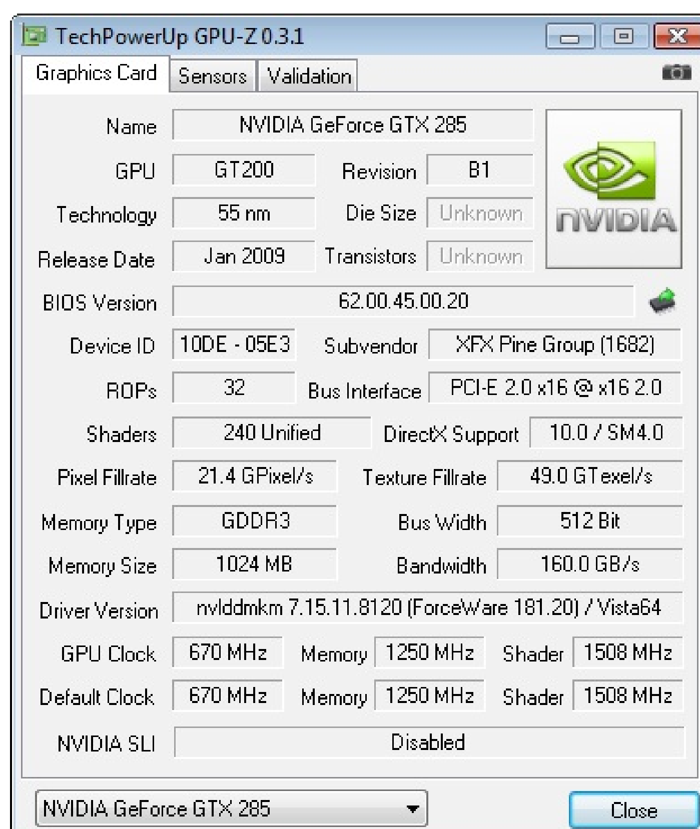


Figura 15: Leitura do programa GPU-Z da GPU GTX285

nas uma pequena parte do algoritmo foi implementado para a GPU. A página na qual foi retirado o código do Karsten Kutza possui um conjunto de códigos de rede neural fonte para entusiastas e pesquisadores (*Japan Singapore AI Centre*), servido de base para estudos. Para a segunda técnica de paralelização, método do neurônio por *thread*, foi criada uma nova rede neural artificial, que faça o mesmo da desenvolvida por Karsten Kutza, mas que todo o treinamento fosse implementado para GPU. A rede neural artificial é de camada simples usada para classificador de padrões, como mostrada na figura 16.

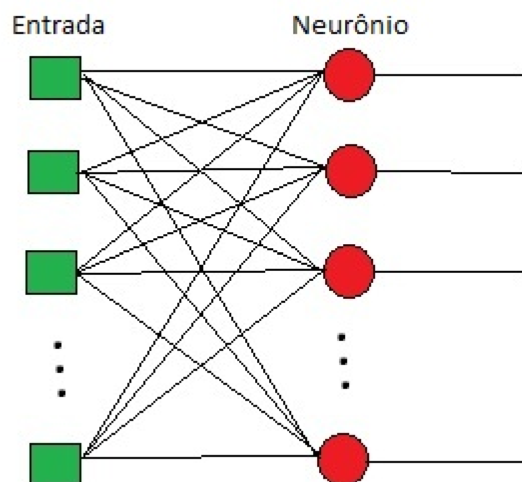


Figura 16: Representação gráfica da rede neural implementada

A rede neural deve ser capaz de reconhecer as classes de representação de dígitos alfa numéricos com dimensões 5x7 como o exemplo do número um na figura 17, também pode ser visto no anexo códigos de RNA.

```
{"  0  "      {-1, -1, 1, -1, -1,
 " 00  "       -1,  1, 1, -1, -1,
 "0 0  "        1, -1, 1, -1, -1,
 "  0  "       -1, -1, 1, -1, -1,
 "  0  "       -1, -1, 1, -1, -1,
 "  0  "       -1, -1, 1, -1, -1,
 "  0  "}      -1, -1, 1, -1, -1}
```

Figura 17: Representação gráfica do dígito do número um

Todos os códigos usados na monografia, incluido o do anexo Crivo de Erastóstenes, foram desenvolvidos primeiramente sobre o *emulation mode*, ou modo de emulação, disponível pelo CUDA até a versão 3.0. O modo de emu-

lação permite ao desenvolvedor emular o ambiente da GPU na CPU, sendo muito útil para encontrar erros nos algoritmos.

O maior benefício oferecido pelo modo de emulação é a possibilidade de usar funções de impressão de variáveis

```
printf()
```

que estejam na *kernel* da GPU, opção que não existe caso o código esteja rodando direto na GPU.

A maior desvantagem do modo de emulação está no número total de *threads* que suporta, apenas 512. Independente do número de *grids* e blocos criados.

O modo de emulação pode ser facilmente usado adicionando a *flag*

```
emu=1
```

após o comando

```
make
```

para compilação do algoritmo.

3.3.1 Método da menor granularidade

No método da menor granularidade somente uma pequena fração do código é paralelizada. Este pequeno trecho geralmente corresponde a parte que possui grande repetição e tempo de computação. A rede neural artificial foi implementada de forma seqüencial para ser executada na CPU. O algoritmo seqüencial é apresentado na figura 18.

Foi identificada a função que mais tempo gasta ao ser executada e esta foi colocada para ser executada na GPU. Para isso, foi utilizada a ferramenta gprof

```
Algoritmo
  Declarações de variáveis e estruturas
  Inicialização das variáveis com valores aleatórios
  Simulação da Rede Neural Artificial
    Propagação de Sinais
    Ajustes de peso
  Fim da simulação
Fim-Algoritmo
```

Figura 18: Algoritmo do código

(Fenlason e Stallman, 2010), que informa o percentual de tempo gasto em cada função.

No código foi identificada a função que mais exige tempo de processamento. A função de nome *AdjustWeights* está dentro da simulação da rede neural. A figura 19 exibe um relatório completo do programa gprof onde pode-se identificar a função anteriormente mencionada gasta 45,83 por cento do tempo total de execução. Esta função é chamada várias vezes em diferentes trechos do programa.

A figura 20 apresenta o algoritmo da função AdjustWeights que será executado de forma seqüencial na CPU.

Utilizando das mesmas técnicas de programação, a função *AdjustWeights* foi adaptada para ser executada na GPU. A figura 21 mostra como ficou o algoritmo adaptado para GPU. Pode-se observar que há uma parte do algoritmo executada na CPU e outra na GPU (*Kernel* GPU)

Na análise de complexidade da função *AdjustWeights* percebe-se que possui uma complexidade de tempo de O^2 devido ao *for* encadeado para ajuste de pe-

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
45.83	0.11	0.11	28400	3.87	3.87	AdjustWeights
41.67	0.21	0.10	31260	3.20	3.20	PropagateNet
8.33	0.23	0.02	31260	0.64	0.64	ComputeOutputError
4.17	0.24	0.01	31260	0.32	0.32	SetInput
0.00	0.24	0.00	31260	0.00	0.00	GetOutput
0.00	0.24	0.00	31260	0.00	7.68	SimulateNet
0.00	0.24	0.00	28400	0.00	0.00	RandomEqualINT
0.00	0.24	0.00	360	0.00	0.00	RandomEqualREAL
0.00	0.24	0.00	10	0.00	0.00	WriteInput
0.00	0.24	0.00	10	0.00	0.00	WriteOutput
0.00	0.24	0.00	1	0.00	0.00	FinalizeApplication
0.00	0.24	0.00	1	0.00	0.00	GenerateNetwork
0.00	0.24	0.00	1	0.00	0.00	InitializeApplication
0.00	0.24	0.00	1	0.00	0.00	InitializeRandoms
0.00	0.24	0.00	1	0.00	0.00	RandomWeights

Figura 19: Relatório gprof

```
Função AdjustWeights()

Declaração de variáveis

Alocação de memória

Preenchimento das variáveis

Para(início de i até tamanho de i)

  Para(início de j até tamanho de j)

    Weight[i][j] += net * Error[i] * Output[j]
```

Figura 20: Função *AdjustWeights* CPU

sos. Utilizando de técnicas de programação específicas e da forma como o CUDA estrutura as threads para serem executadas dentro da GPU é possível reduzir esta complexidade para constante, pois o número de threads utilizadas é igual ao tamanho do vetor de pesos. A figura 22 mostra o algoritmo com a utilização desta técnica. Pode-se observar que na linha 5 a chamada da função *Kernel* é realizada considerando que serão criadas N threads, onde N é o número de elementos do

```
Kernel GPU
Para(início de i até tamanho de i)
  Para(início de j até tamanho de j)
    Weight[i][j] += net * Error[i] * Output[j]


CPU
Função AdjustWeights()
1. Declaração de variáveis
2. Alocação de memória
3. Preenchimento das variáveis
4. Envio das variáveis para a GPU
5. Chamada da função kernel da GPU para ser executada em 1 thread
6. Envio dos dados já processados para a CPU
```

Figura 21: Sequencial GPU (1 *thread*)

vetor de tamanho ixj domo da variável *wheight[i][j]*. O algoritmo do método da menor granularidade pode ser visto no anexo códigos de RNA.

3.3.2 Método do Neurônio por *thread*

A rede neural teve todo o seu treinamento implementado na GPU. Cara thread na GPU apresentará apenas a entrada para o treinamento de um neurônio, como é ilustrado na figura 19.

Como forma de diminuir o número de trocas de informação entre a CPU e GPU foi feito uma análise das variáveis necessárias a todo código e somente ao treinamento. Variáveis importantes somente para o treinamento foram implementadas somente na GPU, livrando a CPU o tempo de criação e a GPU o tempo de cópia da mesma para sua memória.

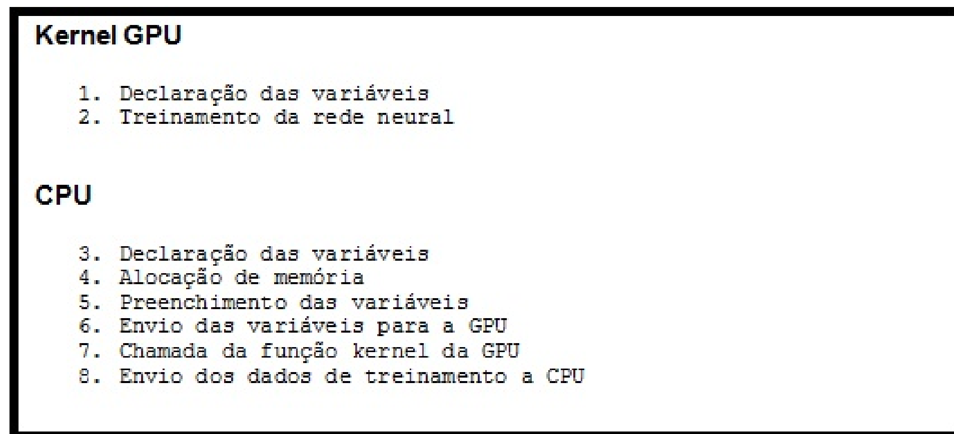
```
Kernel GPU
Weight [idx] += Error[idx - (idx - ( idx/col ))] *
 Output[idx - (col * ( idx/col ))] * net


CPU
Função AdjustWeights()
1. Declaração de variáveis
2. Alocação de memória
3. Preenchimento das variáveis
4. Envio das variáveis para a GPU
5. Chamada da função kernel da GPU para ser executada em N threads
6. Envio dos dados já processados para a CPU
```

Figura 22: Sequencial GPU (N *thread*)

Figura 23: Componentes de uma thread

Com este novo método diminuir drasticamente a troca de informação entre CPU e GPU, reduzindo a apenas 2 momentos, o envio e o retorno do Peso. O algoritmo do método de um neurônio por *thread* pode ser visto no anexo intitulado "‘códigos de RNA"’.



Kernel GPU

1. Declaração das variáveis
2. Treinamento da rede neural

CPU

3. Declaração das variáveis
4. Alocação de memória
5. Preenchimento das variáveis
6. Envio das variáveis para a GPU
7. Chamada da função kernel da GPU
8. Envio dos dados de treinamento a CPU

Figura 24: Algoritmo do método do neurônio por *thread*

4 Resultados

As implementações foram analisadas com diferentes tamanhos do problema: treinamento de 11, 22 e 34 neurônios. Estas quantidades de neurônios foram escolhidas pois apresentaram melhores resultados. Ao tentar dividir o conjunto de 34 neurônios em múltiplo de 17, os resultados estavam inconclusivos. Em múltiplos de 8 neurônios apresentaram um o desperdício de 2 neurônios e em múltiplos de 6 os resultados entre os tamanhos de treinamento ficaram muito próximos.

A comparação de desempenho entre arquiteturas diferentes (CPU e GPU) não é justa, pois temos velocidades de *clock*, quantidade de memória, número de núcleos entre outras variáveis diferentes. Por isso, buscou-se comparar a execução com 1 e com N *threads* na mesma arquitetura (GPU).

Para cada tamanho de problema foram realizados 3 experimentos: execução do código seqüencialmente na CPU, seqüencialmente na GPU utilizando somente uma *thread* e paralelizado na GPU utilizando N *threads*. Cada código foi executado 10 vezes e calculada a média aritmética do tempo gasto. No primeiro tamanho de problema é feito o treinamento da rede neural com 11 neurônios, no segundo são 22 neurônios e no terceiro 34 neurônios.

4.1 Resultados Obtidos com o Método da Menor Granularidade

Para este método, o valor de N é igual a da quantidade de pesos da rede neural artificial. Na figura 25 são apresentados os tempos em microssegundos gastos para a execução dos experimentos com 11 elementos.

Podemos observar na figura 25 que o tempo de execução seqüencial do código na CPU foi menor, aproximadamente 121,9 vezes menor que o mesmo código executado seqüencialmente na GPU e aproximadamente 69.5 menor que o

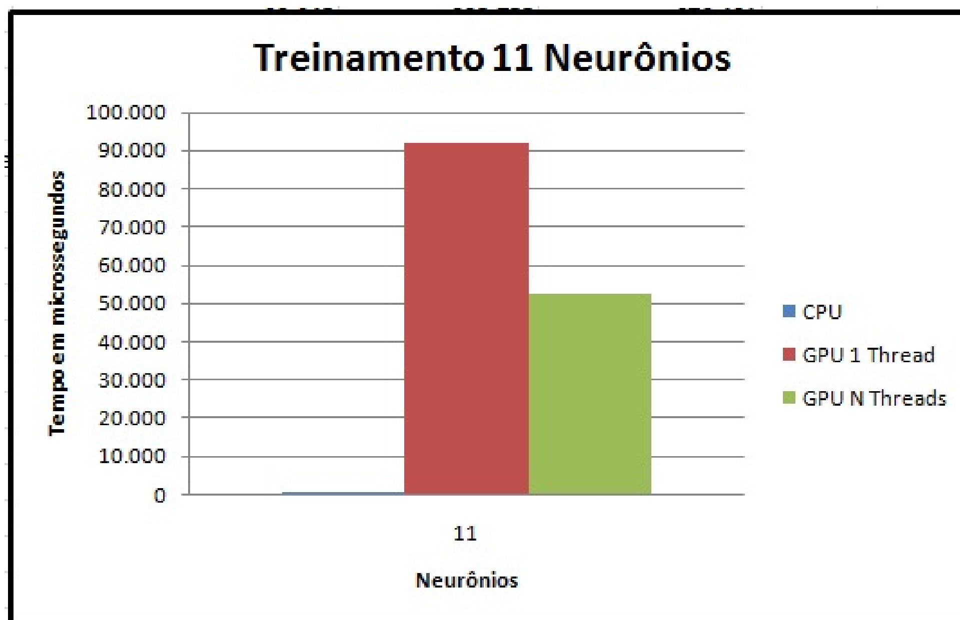


Figura 25: Treinamento 11 neurônios

código usando N *threads* na GPU, isso comparando estruturas de *hardware* diferentes (CPUxGPU).

Comparando o tempo de execução do código na GPU observa-se que quando se utiliza N *threads* obtém-se um tempo de execução em torno de 1,7 vezes menor que o mesmo código executado sequencialmente em uma *thread* na GPU.

Na figura 26 são apresentados os tempos gastos para a execução dos experimentos com 22 elementos.

Podemos observar na figura 26 que o tempo de execução seqüencial do código na CPU foi menor, aproximadamente 112,5 vezes menor que o mesmo código executado seqüencialmente na GPU e aproximadamente 33.5 menor que o código usando N *threads* na GPU, isso comparando estruturas de *hardware* diferentes (CPUxGPU).

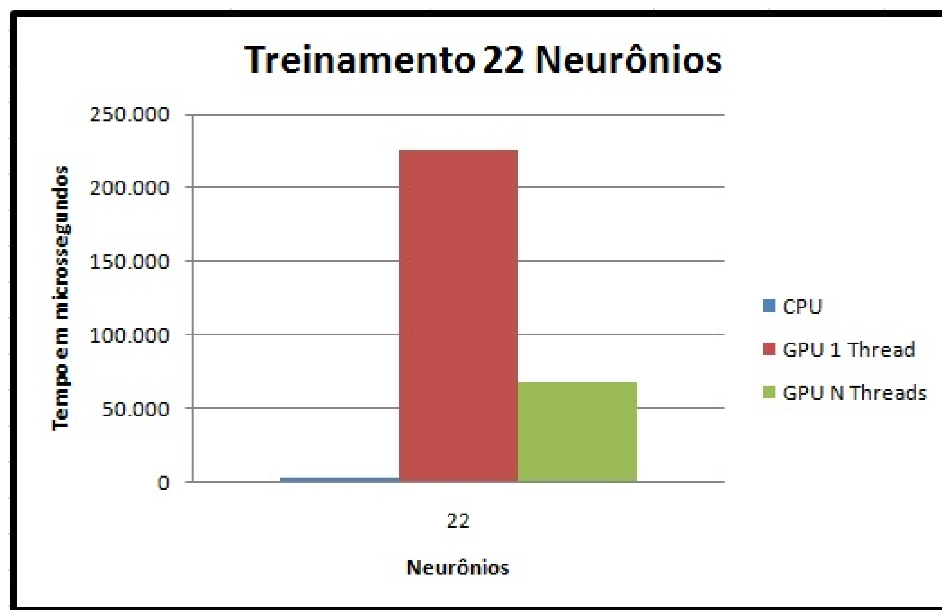


Figura 26: Treinamento 22 neurônios

Comparando o tempo de execução do código na GPU observa-se que quando se utiliza N *threads* obtém-se um tempo de execução em torno de 3,3 vezes menor que o mesmo código executado sequencialmente em uma *thread* na GPU.

Na figura 27 são apresentados os tempos gastos para a execução dos experimentos com 34 elementos.

Podemos observar na figura 27 que o tempo de execução seqüencial do código na CPU foi menor, aproximadamente 112,6 vezes menor que o mesmo código executado seqüencialmente na GPU e aproximadamente 17,8 menor que o código usando N *threads* na GPU, isso comparando estruturas de *hardware* diferentes (CPUxGPU).

Comparando o tempo de execução do código na GPU observa-se que quando se utiliza N *threads* obtém-se um tempo de execução em torno de 6,7

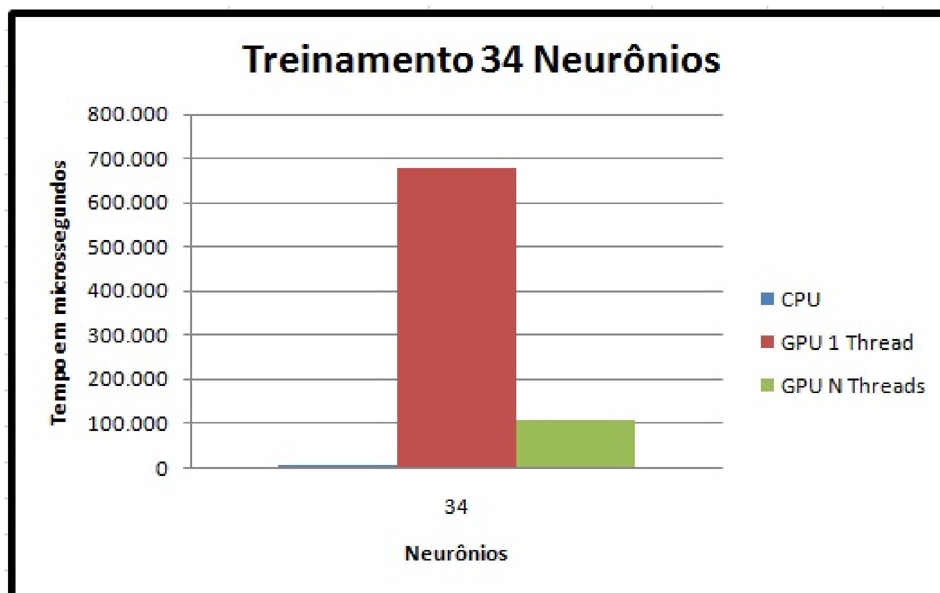


Figura 27: Treinamento 34 neurônios

vezes menor que o mesmo código executado sequencialmente em uma *thread* na GPU.

Observando o aumento do problema e o comportamento entre estruturas de *hardware* e técnicas de programação podemos perceber que o código seqüencial na CPU mesmo sendo mais rápido até então apresentou aumento de tempo de 8,4 vezes, enquanto o aumento do tempo seqüencial na GPU foi de 7,3 vezes e o aumento de tempo utilizando N *threads* na GPU foi de 2,0 vezes.

Dentro de uma mesma arquitetura de *hardware*, podemos notar que a diferença de técnicas de programação faz com que a GPU seja mais bem aproveitada. A diferença de tempo de execução vai se alargando entre as duas implementações apresentadas à medida que o tamanho do problema aumenta.

4.2 Resultados Obtidos com o Método do Neurônio por *Thread*

Para este método, o valor de N é igual a da quantidade de neurônios da rede neural artificial. Na figura 28 são apresentados os tempos em microssegundos gastos para a execução dos experimentos com 11 elementos.

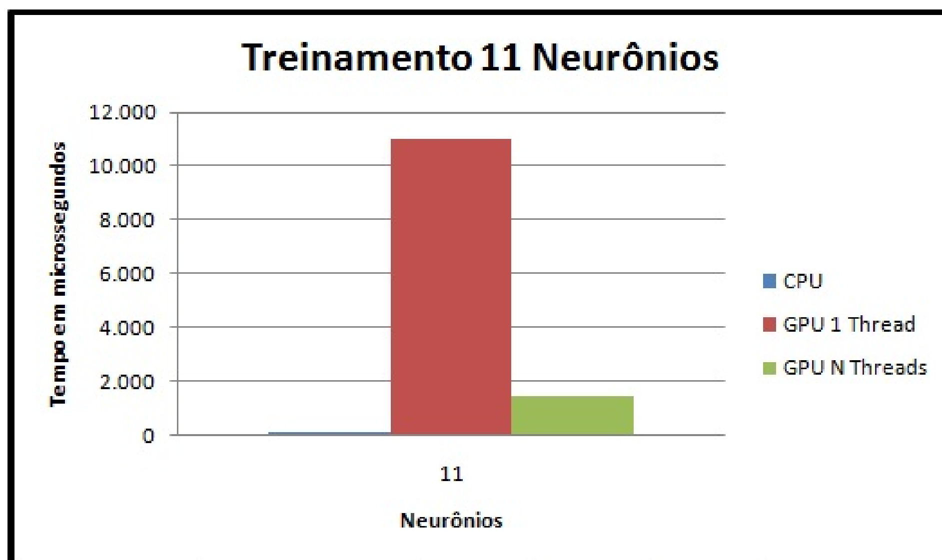


Figura 28: Treinamento 11 neurônios

Podemos observar na figura 28 que o tempo de execução seqüencial do código na CPU foi menor, aproximadamente 523,7 vezes menor que o mesmo código executado seqüencialmente na GPU e aproximadamente 68,7 menor que o código usando N *threads* na GPU, isso comparando estruturas de *hardware* diferentes (CPUxGPU).

Comparando o tempo de execução do código na GPU observa-se que quando se utiliza N *threads* obtém-se um tempo de execução em torno de 7,5 vezes menor que o mesmo código executado sequencialmente em uma *thread* na GPU.

Na figura 29 são apresentados os tempos gastos para a execução dos experimentos com 22 elementos.

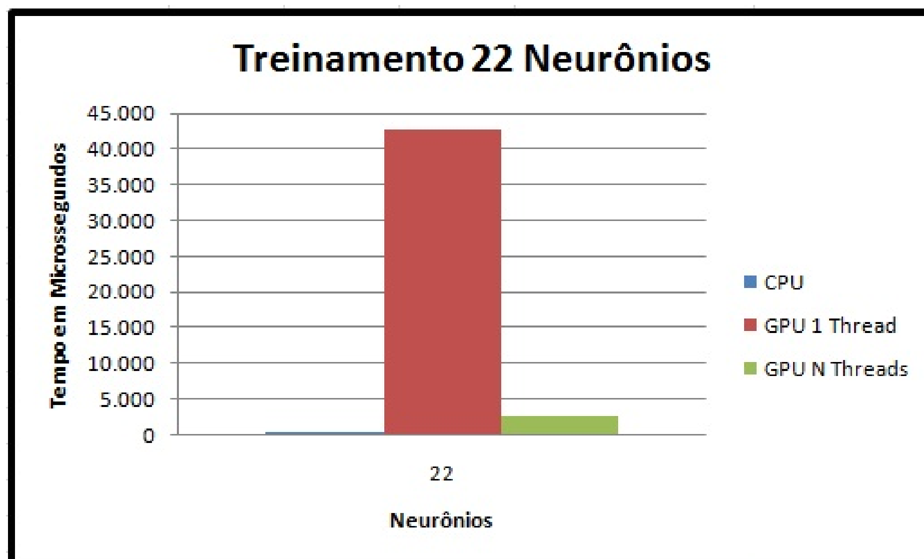


Figura 29: Treinamento 22 neurônios

Podemos observar na figura 29 que o tempo de execução seqüencial do código na CPU foi menor, aproximadamente 577,5 vezes menor que o mesmo código executado seqüencialmente na GPU e aproximadamente 34,3 menor que o código usando N *threads* na GPU, isso comparando estruturas de *hardware* diferentes (CPUxGPU).

Comparando o tempo de execução do código na GPU observa-se que quando se utiliza N *threads* obtém-se um tempo de execução em torno de 17,0 vezes menor que o mesmo código executado sequencialmente em uma *thread* na GPU.

Na figura 30 são apresentados os tempos gastos para a execução dos experimentos com 34 elementos.

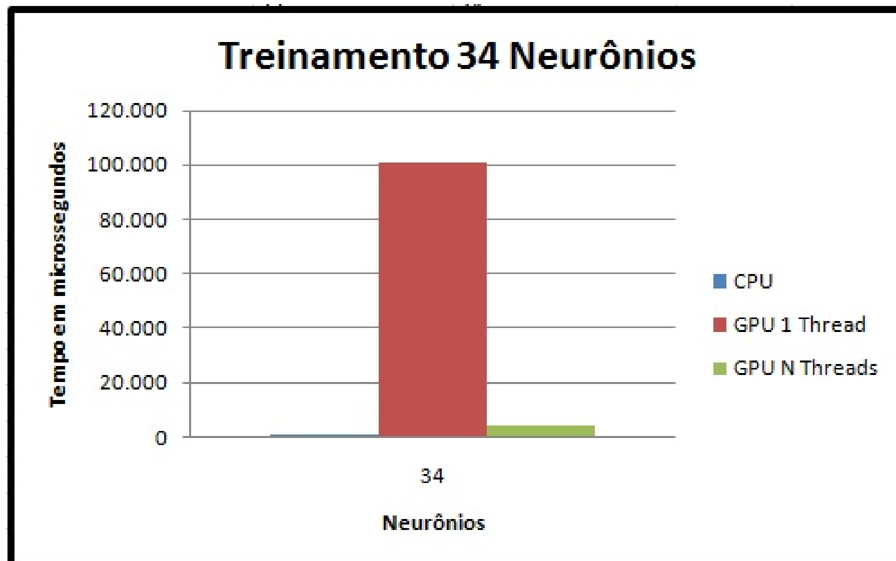


Figura 30: Treinamento 34 neurônios

Podemos observar na figura 30 que o tempo de execução seqüencial do código na CPU foi menor, aproximadamente 538 vezes menor que o mesmo código executado seqüencialmente na GPU e aproximadamente 21,2 menor que o código usando N *threads* na GPU, isso comparando estruturas de *hardware* diferentes (CPUxGPU).

Comparando o tempo de execução do código na GPU observa-se que quando se utiliza N *threads* obtém-se um tempo de execução em torno de 25,2 vezes menor que o mesmo código executado sequencialmente em uma *thread* na GPU.

Observando o aumento do problema, pouco mais de 3 vezes, e o comportamento entre estruturas de *hardware* e técnicas de programação podemos perceber que o código seqüencial na CPU apresentou aumento de tempo em torno de 8,9 vezes. Foi de 2,8 vezes da GPU utilizando *threads* enquanto o aumento do tempo seqüencial na GPU foi de 10,1 vezes.

Dentro de uma mesma arquitetura de *hardware*, podemos notar que a diferença de técnicas de programação faz com que a GPU seja mais bem aproveitada. A diferença de tempo de execução vai se alargando entre as duas implementações apresentadas à medida que o tamanho do problema aumenta.

No anexo Crivo de Erastóstenes será discutido o que foi apontado como gargalo nos métodos de implementação das redes neurais junto a um algoritmo com ganho real de desempenho da GPU sobre a CPU.

5 Conclusões

Este trabalho apresentou duas técnicas diferentes de se implementar e treinar uma rede neural artificial em GPGPU utilizando CUDA: método de menor granularidade e método de um neurônio por *thread*. A mesma implementação também foi realizada em uma CPU para posterior comparação. O tempo de execução de treinamento da rede neural artificial foi menor na CPU quando comparado com as técnicas implementadas na GPGPU.

A implementação do método da menor granularidade teve como desvantagem o número alto de trocas de informação entre a CPU e GPU, que segundo o 'NVIDIA CUDA C *Best Practices Guide*' é algo que deve ser evitado. A implementação do método de um neurônio por *thread* teve como objetivo evitar ao máximo a troca de informação entre a CPU e GPU, mas apresentou como desvantagem a baixa quantidade de *threads* utilizadas, ou seja, tendo em vista que a GPU é um dispositivo de paralelismo em massa, esta foi subutilizada para esta tarefa.

Para melhorar a eficiência, o ideal seria aumentar a quantidade de *threads* no método de um neurônio por *thread*. Os resultados obtidos mostraram que para aproveitar melhor os recursos computacionais de uma GPGPU é necessário utilizar técnicas específicas de programação, designadas exclusivamente para esse tipo de hardware.

5.1 Propostas de Continuidade

Para os trabalhos futuros, sugere-se:

- estudar e implementar novos métodos de implementação de redes neurais artificiais utilizando o CUDA;
- aprofundar o conhecimento sobre programação em GPU utilizando o CUDA;

- estudar o funcionamento do OpenCL;
- testar a rede neural implementada com novos dados de treinamento.
- reconhecimento de padrões em linhas de montagem;
- filtros contra ruídos eletrônicos;

6 Referencia Bibliográfica

AMD Developer Central. **OpenCL and the ATI Stream SDK v2.0** . Disponível em *http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-ATI-Stream-v2.0-Beta.aspx*. Acessado em 29/09/2010.

ARM Ltd and ARM Germany GmbH, 2011. **Sieve of Eratosthenes**. Disponível em *http://www.keil.com/benchmarks/sieve.asp*. Acessado em 17/05/2011.

ATI Stream Technology. **GPU Technology for Accelerated Computing**. Disponível em *http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx*. Acessado em 10/09/2009.

Barreto, J. B. **Introdução às Redes Neurais Artificiais**. 2002. Disponível em *http://www.inf.ufsc.br/ barreto/tutoriais/Survey.pdf*. Acessado em 10/09/2009.

Fang, W., Lau, K. K., Lu, M., Xiao, X., Lam, C. K., Yang, P. Y., He, B., Luo, Q., Sander, P. V. and Yang, K. **Parallel Data Mining on Graphics Processors**. Disponível em *http://gpuminer.googlecode.com/files/gpuminer.pdf*. Acessado em 10/09/2009.

Fenlason J, Stallman R. **GNU gprof: The GNU Profiler**. Disponível em *http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html*. Acessado em 10/09/2009.

Fernandez, A., Martin, R. S., Farguell, E. and Pazienza, G. E. **Cellular Neural Networks Simulation on a Parallel Graphics Processing Unit**. 11th Interna-

tional Workshop on Cellular Neural Networks and their Applications Santiago de Compostela, Spain, 14-16 July 2008.

Foster, I. **Designing and Building Parallel Programs**. Disponível em *http://www.mcs.anl.gov/ itf/dbpp*. Acessado em 10/09/2009.

GPGPU.org. **General-Purpose Computation on Graphics Hardware**. Disponível em *http://gpgpu.org/about*. Acessado em 10/09/2009.

Gschwind, M., Hofstee, P., Flachs, B., Hopkins, M., Watanabe, Y. and Yamazaki, T. Broadband Processor Architecture. **A novel SIMD architecture for the Cell heterogeneous chip-multiprocessor**. Disponível em *http://www.hotchips.org/archives/hc17/2 Mon/HC17.S1/HC17.S1T1.pdf*. Acessado em 10/09/2009.

Hebb, D. O. **The Organization of Behavior: A Neuropsychological Theory**. John Wiley & Sons Inc., 1949.

Jang, H., Park, A., Jung, K. **Neural Network Implementation using CUDA and OpenMP.- Proceedings of the 2008 Digital Image Computing: Techniques and Applications**. Disponível em *http://portal.acm.org/citation.cfm?id=1470322*. Acessado em 10/09/2009.

Japan Singapore AI Centre. **Neural Networks Source Code**. Disponível em *http://tralvex.com/pub/nap/nn-src*. Acessado em 29/09/2010.

Khashman, A. **Neural networks for credit risk evaluation: Investigation of different neural models and learning schemes**. Expert Systems with Applications Volume 37, Issue 9, September 2010, Pages 6233-6239.

Ly, D. L. and Chow, P. **High-Performance Reconfigurable Hardware Architecture for Restricted Boltzmann Machines**. IEEE Transactions on Neural Networks, 2010.

McCulloch, W. and Pitts, W. **A logical calculus of the ideas immanent in nervous activity**. Bulletin of Mathematical Biophysics, vol. 5, num. 4, pages 115-133, 1943.

Marowka, A. **Think Parallel: Teaching Parallel Programming Today**, IEEE Distributed Systems Online, vol. 9, no. 8, 2008, art. no. 0808-o8002. Disponível em *http://www.computer.org/portal/web/csdl/abs/html/mags/ds/2008/08/mds2008 080001.htm*. Acessado em 10/09/2009.

Misra, J. and Saha, I. **Artificial neural networks in hardware: A survey of two decades of progress**. Neurocomputing, Article in Press, 2010.

Mocelin, C. L., Silva, E. F. B., Tobaldini, G. I. **GPU: Aspectos Gerais**. Disponível em *http://www.uri.com.br/ adario/disciplinas/AC2/files/Artigo%20GPU.pdf*. Acessado em 10/09/2009.

Nageswaran, J. M., Dutt, N., Krichmar1, J. L., Nicolau, N., Veidenbaum, A. **Efficient Simulation of Large-Scale Spiking Neural Networks Using CUDA Graphics Processors**. Disponível em *http://www.ics.uci.edu/ jmoorkan/pub/gpusnn-ijcnn.pdf*. Acessado em 10/09/2009.

NVIDIA Corporation. **CUDA C Best Practices Guide**. Disponível em *http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf*. Acessado em 10/09/2009.

NVIDIA Corporation. **Programing Guide 3.0**. Disponível em *http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf*. Acessado em 10/09/2009.

NVIDIA Corporation. **What is Cuda?**. Disponível em *http://www.nvidia.com/object/cuda_what_is.html*. Acessado em 10/09/2009.

Pamplona, Vitor. **Análise de Performance: C vs CUDA**. Disponível em *http://vitorpamplona.com/wiki/An%C3%A1lise%20de%20Performance:%20C%20vs%20Cuda*. Acessado em 10/09/2009.

Poli, G., Saito, J. H., Mari, J. F. and Zorzan, M. R. **Processing Neocognitron of Face Recognition on High Performance Environment Based on GPU with CUDA Architecture**. 20th International Symposium on Computer Architecture and High Performance Computing, 2008.

Rosenblatt, F. **Principles of Neurodynamics. Perceptrons and the Theory of Brain Machanisms**. Cornell Aeronautical Laboratory, 1961.

Scanzio, S. , Cumani, S. , Gemello, R. , Man, F. and Laface, P. **Parallel implementation of Artificial Neural Network training for speech recognition**. Pattern Recognition Letters. Volume 31, Issue 11, 1 August 2010, Pages 1302-1309.

Shi, M., Pan, W., Garis, H. and Chen, K. **Approach to Controlling Robot by Artificial Brain Based on Parallel Evolutionary Neural Network**. 2nd International Conference on Industrial Mechatronics and Automation, 30-31 May, China, 2010.

Tatibana, C. Y. , Kaetsu, D. **Aplicações de Redes Neurais Artificiais 2009** Disponível em *http://www.din.uem.br/ia/neurais*. Acessado em 10/09/2009.

7 ANEXO Crivo de Erastóstenes

O Crivo de Eratóstenes(ARM Ltd, 2011) é um algotirmo padrão utilizado para determinar a velocidade relativa de diferentes computadores, ou, neste caso, a eficiência do código gerado para a CPU e GPU. O algoritmo foi desenvolvido peneira na Grécia antiga e é um dos vários métodos usados para encontrar números primos.

A fim de testar o real poder de processamento paralelo da GPU, foram implementados 2 algoritmos do Crivo de Eratóstenes, um seqüencial para CPU e um paralelo para GPU.

O processo é:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Começando após 2, eliminar todos os múltiplos de 2.

2 3 5 7 9 11 13 15 17 19 21 23 25

Começando após 3, eliminar todos os múltiplos de 3.

2 3 5 7 11 13 17 19 23 25

Começando após 5, eliminar todos os múltiplos de 5.

2 3 5 7 11 13 17 19 23

Continue até o próximo número ainda é maior do que a raiz quadrada do maior número na série original. Neste caso, o próximo número, 7, é maior do que a raiz quadrada de 25, portanto, o processo pára. Os números restantes são primos.

2 3 5 7 11 13 17 19 23

Mas para esta medição de performance, afim de aumentar o número de processamento nas plataformas, a verificação para o próximo número primo continuará até a metade do maior número da série.

Para o algoritmo do Crivo de Eratóstenes que será executado junto a GPU, foram seguidos da melhor maneira possível as informações contidas no NVIDIA

CUDA C *Best Practices Guide*. Entre elas, maior número de *threads* possíveis, menor número de processamento por *thread*, e menor número de troca de informação entre GPU e CPU. Tanto que a série utilizada será gerada na própia GPU, e número de elementos na série será igual ao número de *threads* criadas.

Figura 31: Tempo do Crivo

Observando a figura 31, podemos concluir que com o aumento da série para números próximos a 10^6 o Crivo de Erastóstenes na GPU foi 16,5 vezes mais rápido que na CPU.

8 ANEXO Código Crivo de Eratóstenes

Código Crivo de Eratóstenes para GPU com 10^6 *threads*.

```
#include <cuda.h>
#include <cutil.h>
#include<stdio.h>
#include <cuda_runtime.h>



////////////////////////////////////////////////////////////////////////////////////
//GPU KERNEL

__shared__ int k;
__global__ void crivo_array(int *a)

{

int idx = blockIdx.x*blockDim.x+threadIdx.x;
a[idx]= idx;
k=2;


while(k*2<=idx){

if (((a[idx] % k) == (0))&&(a[idx] > 0)&&(a[idx] != k)){
```

```
a[idx] = a[idx] * -1;
}

k++;

while(a[k]<0){
k++;
}

}

}

////////////////////////////////////////////////////////////////////////////////////
//CPU

 int main(void)
{

int *a_h, *a_d, threads, blocks;

const int N=1000000;

if (N < 256)
{
threads = N;
```

```
blocks = 1;
}
else
{
threads = 256;
if ((N % threads) == 0)
blocks = N/threads;
else
blocks = (N/threads) + 1;
}

size_t size = N*sizeof(int);

a_h=(int*)malloc(size);

cudaMalloc((void**)&a_d,size);

unsigned int timer = 0;
CUT_SAFE_CALL(cutCreateTimer(&timer));
CUT_SAFE_CALL(cutStartTimer(timer));

cudaMemcpy(a_d,a_h,size,cudaMemcpyHostToDevice);
crivo_array<<<blocks,threads>>>(a_d);
cudaMemcpy(a_h,a_d,size,cudaMemcpyDeviceToHost);

printf("\nTempo de computacao: %f (ms) \n\n", cutGetTimerValue(timer));
```

```
CUT_SAFE_CALL(cutDeleteTimer(timer));

a_h[1] = -1;
printf("SÃ£o primos os numeros entre 2 e %d\n", N);
for(int i=0; i<N; i++)
if (a_h[i] > 0)
printf("%d\n",a_h[i]);



free(a_h);
cudaFree(a_d);

}
```

Código Crivo de Eratóstenes para CPU com a série de 10^6 elementos.

```
#include <cuda.h>
#include <cutil.h>
#include<stdio.h>
#include <cuda_runtime.h>

 int main(void)
{
int *a_h, i, j;

j=0;
```

```
const int N=1000000;

size_t size = N*sizeof(int);

a_h=(int*)malloc(size);

for( i=0; i<N; i++)
a_h[i]= i + 2;

int k = a_h[0];

unsigned int timer = 0;
CUT_SAFE_CALL(cutCreateTimer(&timer));
CUT_SAFE_CALL(cutStartTimer(timer));

while (j*2 <= (N-2))
{

j++;
for(i=j; i<N; i++){
if (((a_h[i] % k) == (0))&&(a_h[i] > 0))
a_h[i] = a_h[i] * -1;
}

while (a_h[j] < N)
```

```
{
j++;
if (a_h[j] > 0)
break;
}

k = a_h[j];
}

printf("\nTempo de computacao: %f (ms) \n\n", cutGetTimerValue(timer));
CUT_SAFE_CALL(cutDeleteTimer(timer));

printf("SÃ£o primos os numeros entre 2 e %d\n", N);
for(i=0; i<(N-2); i++)
if (a_h[i] > 0)
printf("%d\n",a_h[i]);


}
```

9 ANEXO Códigos de RNA

Código do método da menor granularidade com somente 11 classes e N *threads*.

```
#include <stdlib.h>
#include <stdio.h>
#include <cuda_runtime.h>
#include <cutil.h>



typedef int           BOOL;
typedef char          CHAR;
typedef int           INT;
typedef float        REAL; //typedef double        REAL;

#define FALSE         0
#define TRUE          1
#define NOT           !
#define AND           &&
#define OR            ||

#define MIN(x,y)      ((x)<(y) ? (x) : (y))
#define MAX(x,y)      ((x)>(y) ? (x) : (y))

#define LO            -1
#define HI            +1
```

```
#define BIAS           1

#define sqr(x)         ((x)*(x))


//variavel global
   REAL *vetorA ;
   INT *vetorB ;
   REAL *matrizAB;
   REAL *d_vetorA ;
   INT *d_vetorB;
   REAL *d_matrizAB ;
   REAL *SUPERmatrizAB ;/// Net->OutputLayer->Weight[i][j]
   size_t size = sizeof(float) * 11 * 36;


typedef struct {                     /* A LAYER OF A NET:                     */
        INT           Units;         /* - number of units in this layer       */
        REAL*         Activation;    /* - activation of ith unit              */
        INT*          Output;        /* - output of ith unit                  */
        REAL*         Error;         /* - error term of ith unit              */
        REAL**        Weight;        /* - connection weights to ith unit      */
} LAYER;

typedef struct {                     /* A NET:                                */
        LAYER*        InputLayer;    /* - input layer                         */
```

```
        LAYER*        OutputLayer;   /* - output layer                        */
        REAL          Eta;           /* - learning rate                       */
        REAL          Error;         /* - total net error                     */
        REAL          Epsilon;       /* - net error to terminate training     */
} NET;


/******************************************************************************
        R A N D O M S   D R A W N   F R O M   D I S T R I B U T I O N S
 ******************************************************************************/


void InitializeRandoms()
{
  srand(4711);
}


INT RandomEqualINT(INT Low, INT High)
{
  return rand() % (High-Low+1) + Low;
}


REAL RandomEqualREAL(REAL Low, REAL High)
{
```

```
  return ((REAL) rand() / RAND_MAX) * (High-Low) + Low;
}


/******************************************************************************
              A P P L I C A T I O N - S P E C I F I C   C O D E
 ******************************************************************************/


#define NUM_DATA      11
#define X             5
#define Y             7

#define N             (X * Y)
#define M             11

//Pattern[NUM_DATA][X][Y]
CHAR                  Pattern[NUM_DATA][7][6] = { { " OOO ",
                                                    "O   O",
                                                    "O   O",
                                                    "O   O",
                                                    "O   O",
                                                    "O   O",
                                                    " OOO "  },

                                                  { "  O  ",
```

```
  " OO  ",
  "O O  ",
  "  O  ",
  "  O  ",
  "  O  ",
  "  O  "  },

{ " OOO ",
  "O   O",
  "    O",
  "   O ",
  "  O  ",
  " O   ",
  "OOOOO"  },

{ " OOO ",
  "O   O",
  "    O",
  " OOO ",
  "    O",
  "O   O",
  " OOO "  },

{ "   O ",
  "  OO ",
  " O O ",
```

```
  "O  O ",
  "OOOOO",
  "   O ",
  "   O "  },

{ "OOOOO",
  "O    ",
  "O    ",
  "OOOO ",
  "    O",
  "O   O",
  " OOO "  },

{ " OOO ",
  "O   O",
  "O    ",
  "OOOO ",
  "O   O",
  "O   O",
  " OOO "  },

{ "OOOOO",
  "    O",
  "    O",
  "   O ",
  "  O  ",
```

```
  " O   ",
  "O    "  },

{ " OOO ",
  "O   O",
  "O   O",
  " OOO ",
  "O   O",
  "O   O",
  " OOO "  },

{ " OOO ",
  "O   O",
  "O   O",
  " OOOO",
  "    O",
  "O   O",
  " OOO "  },

{ "  O  ",
  " O O ",
  "O   O",
  "O   O",
  "OOOOO",
  "O   O",
  "O   O"  } };
```

```
INT                    Input [NUM_DATA][N];
INT                    Output[NUM_DATA][M] =

            { {HI, LO, LO, LO, LO, LO, LO, LO, LO, LO, LO},
              {LO, HI, LO, LO, LO, LO, LO, LO, LO, LO, LO},
              {LO, LO, HI, LO, LO, LO, LO, LO, LO, LO, LO},
              {LO, LO, LO, HI, LO, LO, LO, LO, LO, LO, LO},
              {LO, LO, LO, LO, HI, LO, LO, LO, LO, LO, LO},
              {LO, LO, LO, LO, LO, HI, LO, LO, LO, LO, LO},
              {LO, LO, LO, LO, LO, LO, HI, LO, LO, LO, LO},
              {LO, LO, LO, LO, LO, LO, LO, HI, LO, LO, LO},
              {LO, LO, LO, LO, LO, LO, LO, LO, HI, LO, LO},
              {LO, LO, LO, LO, LO, LO, LO, LO, LO, HI, LO},
              {LO, LO, LO, LO, LO, LO, LO, LO, LO, LO, HI}  };

FILE*                  f;


void InitializeApplication(NET* Net)
{
  INT n,i,j;

  Net->Eta     = 0.001;
  Net->Epsilon = 0.0001;
```

```
  for (n=0; n<NUM_DATA; n++) {
    for (i=0; i<Y; i++) {
      for (j=0; j<X; j++) {
        Input[n][i*X+j] = (Pattern[n][i][j] == 'O') ? HI : LO;
      }
    }
  }
  f = fopen("SAIDA.txt", "w");
}


void WriteInput(NET* Net, INT* Input)
{
  INT i;

  for (i=0; i<N; i++) {
    if (i%X == 0) {
      fprintf(f, "\n");
    }
    fprintf(f, "%c", (Input[i] == HI) ? 'O' : ' ');
  }
  fprintf(f, " -> ");
}


void WriteOutput(NET* Net, INT* Output)
```

```
{
  INT i;
  INT Count, Index;

  Count = 0;
  for (i=0; i<M; i++) {
    if (Output[i] == HI) {
      Count++;
      Index = i;
    }
  }
  if (Count == 1)
    fprintf(f, "%i\n", Index);
  else
    fprintf(f, "%s\n", "invalid");
}


void FinalizeApplication(NET* Net)
{
  fclose(f);
}


/******************************************************************************
                              I N I T I A L I Z A T I O N
```

```
******************************************************************************/


void GenerateNetwork(NET* Net)
{
  INT i;

  Net->InputLayer  = (LAYER*) malloc(sizeof(LAYER));
  Net->OutputLayer = (LAYER*) malloc(sizeof(LAYER));

  Net->InputLayer->Units       = N;
  Net->InputLayer->Output      = (INT*)   calloc(N+1, sizeof(INT));
  Net->InputLayer->Output[0]   = BIAS;

  Net->OutputLayer->Units      = M;
  Net->OutputLayer->Activation = (REAL*)  calloc(M+1, sizeof(REAL));
  Net->OutputLayer->Output     = (INT*)   calloc(M+1, sizeof(INT));
  Net->OutputLayer->Error      = (REAL*)  calloc(M+1, sizeof(REAL));
  Net->OutputLayer->Weight     = (REAL**) calloc(M+1, sizeof(REAL*));

  for (i=1; i<=M; i++) {
    Net->OutputLayer->Weight[i] = (REAL*) calloc(N+1, sizeof(REAL));
  }

  Net->Eta     = 0.1;
  Net->Epsilon = 0.01;
```

```
}


void RandomWeights(NET* Net)
{
  INT i,j,k;

  k=0;
  for (i=1; i<=Net->OutputLayer->Units; i++) {
    for (j=0; j<=Net->InputLayer->Units; j++) {
      SUPERmatrizAB[k] = 0;
      k++;
    }
  }
}


void SetInput(NET* Net, INT* Input, BOOL Protocoling)
{
  INT i;

  for (i=1; i<=Net->InputLayer->Units; i++) {
    Net->InputLayer->Output[i] = Input[i-1];
    vetorB[i] = Input[i-1];
  }
  if (Protocoling) {
```

```
    WriteInput(Net, Input);
  }
}


void GetOutput(NET* Net, INT* Output, BOOL Protocoling)
{
  INT i;

  for (i=1; i<=Net->OutputLayer->Units; i++) {
    Output[i-1] = Net->OutputLayer->Output[i];
  }
  if (Protocoling) {
    WriteOutput(Net, Output);
  }
}


/******************************************************************************
                    P R O P A G A T I N G   S I G N A L S
 ******************************************************************************/


void PropagateNet(NET* Net)
{
  INT  i,j,k;
```

```
  REAL Sum;

  k=0;
  for (i=1; i<=Net->OutputLayer->Units; i++) {
    Sum = 0;
    for (j=0; j<=Net->InputLayer->Units; j++) {
      Sum += SUPERmatrizAB[k] * Net->InputLayer->Output[j];
      k++;
    }
    Net->OutputLayer->Activation[i] = Sum;
    if (Sum >= 0)
      Net->OutputLayer->Output[i] = HI;
    else
      Net->OutputLayer->Output[i] = LO;
  }
}


/******************************************************************************
                       A D J U S T I N G   W E I G H T S
 ******************************************************************************/


void ComputeOutputError(NET* Net, INT* Target)
{
  INT  i;
```

```
  REAL Err;

  Net->Error = 0;
  for (i=1; i<=Net->OutputLayer->Units; i++) {
    Err = Target[i-1] - Net->OutputLayer->Output[i];
    Net->OutputLayer->Error[i] = Err;
    vetorA[i-1] = Err; // NEW
    Net->Error += 0.5 * sqr(Err);
  }
}

//////////////////////////////////////////////////////////////////////
//Kernel GPU

__global__ void multi_vetor( REAL *A, INT *B, REAL *AB, INT col, REAL net )

{

   int idx = threadIdx.x ;

   AB[idx] += A[idx - (idx - ( idx/col ))] * B[idx - (col * ( idx/col ))] * net;

}

////////////////////////////////////////////////////////////////////////
```

```
void AdjustWeights(NET* Net) //Esta funÃ§Ã£o deve ser paralelizada no CUDA
{

  REAL net;
  int col;

   // Codigo CUDA
   cudaMemcpy(d_vetorA, vetorA, size, cudaMemcpyHostToDevice);
   cudaMemcpy(d_vetorB, vetorB, size, cudaMemcpyHostToDevice);
   cudaMemcpy(d_matrizAB, SUPERmatrizAB, size, cudaMemcpyHostToDevice);

   net = Net->Eta;
   col = 36;
   multi_vetor<<<1, 396>>>(d_vetorA, d_vetorB, d_matrizAB, col, net);

   cudaMemcpy(SUPERmatrizAB, d_matrizAB, size, cudaMemcpyDeviceToHost);

}

//////////////////////////////////////////////////////////////////////////////
/******************************************************************************
                      S I M U L A T I N G   T H E   N E T
 ******************************************************************************/


void SimulateNet(NET* Net, INT* Input, INT* Target, BOOL Training,
```

```
 BOOL Protocoling)
{
  INT Output[M];

  SetInput(Net, Input, Protocoling);
  PropagateNet(Net);
  GetOutput(Net, Output, Protocoling);

  ComputeOutputError(Net, Target);
  if (Training)
    AdjustWeights(Net);
}


/******************************************************************************
                                    M A I N
 ******************************************************************************/


int main()
{
  NET  Net;
  REAL Error;
  BOOL Stop;
  INT  n,m;
```

```
unsigned int timer = 0;
cutCreateTimer(&timer);
cutStartTimer(timer);


vetorA= (REAL*) malloc(size);
vetorB= (int*) malloc(size);
matrizAB= (REAL*) malloc(size);
SUPERmatrizAB= (REAL*) malloc(size);

cudaMalloc((void**)&d_vetorA, size);
cudaMalloc((void**)&d_vetorB, size);
cudaMalloc((void**)&d_matrizAB, size);


InitializeRandoms();
GenerateNetwork(&Net);
RandomWeights(&Net);
InitializeApplication(&Net);




do {
  Error = 0;
```

```
  Stop = TRUE;
  for (n=0; n<NUM_DATA; n++) {
    SimulateNet(&Net, Input[n], Output[n], FALSE, FALSE);
    Error = MAX(Error, Net.Error);
    Stop = Stop AND (Net.Error < Net.Epsilon);
  }
  Error = MAX(Error, Net.Epsilon);
  if (NOT Stop) {
    for (m=0; m<10*NUM_DATA; m++) {
      n = RandomEqualINT(0, NUM_DATA-1);
      SimulateNet(&Net, Input[n], Output[n], TRUE, FALSE);
    }
  }
} while (NOT Stop);

for (n=0; n<NUM_DATA; n++) {
  SimulateNet(&Net, Input[n], Output[n], FALSE, TRUE);
}

printf("\n Tempo de computacao: %f (ms) \n", cutGetTimerValue(timer));
cutDeleteTimer(timer);

FinalizeApplication(&Net);

cudaFree(d_vetorA);
cudaFree(d_vetorB);
```

```
  cudaFree(d_matrizAB);

}
```

Código do método do neurônio por *thread* com apenas 11 neurônios e cada neurônio em uma *thread*.

```
#include <stdlib.h>
#include <stdio.h>
#include <cuda_runtime.h>
#include <cutil.h>


FILE*                 f;


//////////////////////////// //CUDA// ////////////////////////////

__global__ void treinamento(float *Peso){


int idx = blockIdx.x*blockDim.x+threadIdx.x;

float NetError, Soma, Ativa[11], ERRO[11], Xin[36];
int  n, i, j, conta, Parada, YSaida[11];
```

```
float Alfa     = 0.001;
float tolerancia = 0.0001;

int XQuantidade      = 35;
int YinQuantidade     = 11;

Xin[0] = 1;


float Entrada [11][35] = {{-1, 1, 1, 1, -1,
                         1, -1, -1, -1, 1,
                         1, -1, -1, -1, 1,
                         1, -1, -1, -1, 1,
                         1, -1, -1, -1, 1,
                         1, -1, -1, -1, 1,
                         -1, 1, 1, 1, -1 },

                         {-1, -1, 1, -1, -1,
                          -1, -1, 1, -1, -1,
                          -1, -1, 1, -1, -1,
                          -1, -1, 1, -1, -1,
                          -1, -1, 1, -1, -1,
                          -1, -1, 1, -1, -1,
                          -1, -1, 1, -1, -1 },

                          {-1, 1, 1, 1, -1,
```

```
 1, -1, -1, -1, 1,
 -1, -1, -1, -1, 1,
 -1, 1, 1, 1, -1,
 1, -1, -1, -1, -1,
 1, -1, -1, -1, -1,
 -1, 1, 1, 1, 1 },

{1, 1, 1, 1, -1,
 -1, -1, -1, -1, 1,
 -1, -1, -1, -1, 1,
 -1, 1, 1, 1, -1,
 -1, -1, -1, -1, 1,
 -1, -1, -1, -1, 1,
 1, 1, 1, 1, -1 },

{1, -1, -1, -1, 1,
 1, -1, -1, -1, 1,
 1, -1, -1, -1, 1,
 -1, 1, 1, 1, 1,
 -1, -1, -1, -1, 1,
 -1, -1, -1, -1, 1,
 -1, -1, -1, -1, 1 },

{-1, 1, 1, 1, 1,
 1, -1, -1, -1, -1,
 1, -1, -1, -1, -1,
```

```
-1, 1, 1, 1, -1,
-1, -1, -1, -1, 1,
-1, -1, -1, -1, 1,
1, 1, 1, 1, -1 },

{-1, -1, 1, 1, 1,
-1, 1, -1, -1, -1,
1, -1, -1, -1, -1,
1, 1, 1, 1, -1,
1, -1, -1, -1, 1,
1, -1, -1, -1, 1,
-1, 1, 1, 1, -1 },

{1, 1, 1, 1, 1,
-1, -1, -1, -1, 1,
-1, -1, -1, 1, -1,
-1, -1, 1, -1, -1,
-1, -1, 1, -1, -1,
-1, -1, 1, -1, -1,
-1, -1, 1, -1, -1 },

{-1, 1, 1, 1, -1,
1, -1, -1, -1, 1,
1, -1, -1, -1, 1,
-1, 1, 1, 1, -1,
1, -1, -1, -1, 1,
```

```
                    1, -1, -1, -1, 1,
                    -1, 1, 1, 1, -1 },

                    {-1, 1, 1, 1, -1,
                    1, -1, -1, -1, 1,
                    1, -1, -1, -1, 1,
                    -1, 1, 1, 1, 1,
                    -1, -1, -1, -1, 1,
                    -1, -1, -1, -1, 1,
                    1, 1, 1, 1, -1 },

                    {-1, -1, 1, -1, -1, //A
                    -1, 1, -1, 1, -1,
                    1, -1, -1, -1, 1,
                    1, -1, -1, -1, 1,
                    1, 1, 1, 1, 1,
                    1, -1, -1, -1, 1,
                    1, -1, -1, -1, 1 }};

float YDesejado[11][11] = {{1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{-1, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{-1, -1, 1, -1, -1, -1, -1, -1, -1, -1, -1},
{-1, -1, -1, 1, -1, -1, -1, -1, -1, -1, -1},
{-1, -1, -1, -1, 1, -1, -1, -1, -1, -1, -1},
{-1, -1, -1, -1, -1, 1, -1, -1, -1, -1, -1},
{-1, -1, -1, -1, -1, -1, 1, -1, -1, -1, -1},
```

```
{-1, -1, -1, -1, -1, -1, -1, 1, -1, -1, -1},
{-1, -1, -1, -1, -1, -1, -1, -1, 1, -1, -1},
{-1, -1, -1, -1, -1, -1, -1, -1, -1, 1, -1},
{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 1}};

n=idx;

for (i=1; i<=XQuantidade; i++) {
Xin[i] = Entrada[n][i-1];
}

Parada = 0;

while (Parada != 1) {

Parada = 1;



conta= 396 * idx;
for (i=1; i<=YinQuantidade; i++) {
Soma = 0;
for (j=0; j<=XQuantidade; j++) {
Soma += __fmul_rn(Peso[conta], Xin[j]);
conta++;
}
Ativa[i] = Soma;
```

```
if (Soma >= 0)
YSaida[i] = 1;
else
YSaida[i] = -1;

}

NetError = 0;
for (i=1; i<=YinQuantidade; i++) {
ERRO[i] = YDesejado[n][i-1] - YSaida[i];
NetError += __fmul_rn(ERRO[i], ERRO[i]);
}

conta=396 * idx;
for (i=1; i<=YinQuantidade; i++) {
for (j=0; j<=XQuantidade; j++) {
Peso[conta] += __fmul_rn(__fmul_rn(Alfa, ERRO[i]), Xin[j]);
conta++;
}
}


if (NetError > tolerancia)
Parada = 0;
```

```
}

}

///////////////////////////   Fim CUDA  ///////////////////////////

///////////////////////////   RANDOM  ///////////////////////////

float RandomFloat(float Low, float High)
{
  return ((float) rand() / RAND_MAX) * (High-Low) + Low;
}

///////////////////////////   Fim RANDOM  ///////////////////////////

/////////////////////////// //Yperceptron// ///////////////////////////
```

```
void yperceptron(float* ERRO, float* Peso, float* Ativa, int* Xin,
int* YSaida, int XQuantidade, int YinQuantidade, int* Entrada,
 int* Target, int n)
{

int YDesejado2[11];
int i;

for (i=1; i<=XQuantidade; i++) {
Xin[i] = Entrada[i-1];
}



for (i=0; i<35; i++) {
if (i%5 == 0) {
fprintf(f, "\n");
}
fprintf(f, "%c", (Entrada[i] == 1) ? 'O' : ' ');
}
fprintf(f, " -> ");



int  j, conta;
float Soma;

conta= 396 * n;
```

```
for (i=1; i<=YinQuantidade; i++) {
Soma = 0;
for (j=0; j<=XQuantidade; j++) {
Soma += Peso[conta] * Xin[j];
conta++;
}
Ativa[i] = Soma;
if (Soma >= 0)
YSaida[i] = 1;
else
YSaida[i] = 0;
}


for (i=1; i<=YinQuantidade; i++) {
YDesejado2[i-1] = YSaida[i];
//printf("YSaida[%d] %d\n", i, YSaida[i]);
}


int Count, Index;

Count = 0;
for (i=0; i<11; i++) {
if (YDesejado2[i] == 1) {
Count++;
```

```
Index = i;
}
}
if (Count == 1)
fprintf(f, "%i\n", Index);
else
fprintf(f, "%s\n", "invalid");


}


//////////////////////////// //MAIN// ////////////////////////////


int main()
{

float Ativa[11], ERRO[11];
int  n, conta, YSaida[11], Xin[36];


int XQuantidade = 35;
int YinQuantidade = 11;
```

```
Xin[0] = 1;
srand(4711);



size_t size = 4356*sizeof(float);
float* Peso, *_Peso;
Peso        =(float*)malloc(size);

cudaMalloc((void**)&_Peso,size);


for (conta=0; conta<=4356; conta++) {
Peso[conta]= 0;
}



int Entrada [11][35] = {{-1, 1, 1, 1, -1,
                         1, -1, -1, -1, 1,
                         1, -1, -1, -1, 1,
                         1, -1, -1, -1, 1,
                         1, -1, -1, -1, 1,
                         1, -1, -1, -1, 1,
                         -1, 1, 1, 1, -1 },
```

```
{-1, -1, 1, -1, -1,
 -1, -1, 1, -1, -1,
 -1, -1, 1, -1, -1,
 -1, -1, 1, -1, -1,
 -1, -1, 1, -1, -1,
 -1, -1, 1, -1, -1,
 -1, -1, 1, -1, -1 },

 {-1, 1, 1, 1, -1,
 1, -1, -1, -1, 1,
 -1, -1, -1, -1, 1,
 -1, 1, 1, 1, -1,
 1, -1, -1, -1, -1,
 1, -1, -1, -1, -1,
 -1, 1, 1, 1, 1 },

{1, 1, 1, 1, -1,
 -1, -1, -1, -1, 1,
 -1, -1, -1, -1, 1,
 -1, 1, 1, 1, -1,
 -1, -1, -1, -1, 1,
 -1, -1, -1, -1, 1,
 1, 1, 1, 1, -1 },

{1, -1, -1, -1, 1,
 1, -1, -1, -1, 1,
```

```
 1, -1, -1, -1, 1,
 -1, 1, 1, 1, 1,
 -1, -1, -1, -1, 1,
 -1, -1, -1, -1, 1,
 -1, -1, -1, -1, 1 },

{-1, 1, 1, 1, 1,
 1, -1, -1, -1, -1,
 1, -1, -1, -1, -1,
 -1, 1, 1, 1, -1,
 -1, -1, -1, -1, 1,
 -1, -1, -1, -1, 1,
 1, 1, 1, 1, -1 },

 {-1, -1, 1, 1, 1,
 -1, 1, -1, -1, -1,
 1, -1, -1, -1, -1,
 1, 1, 1, 1, -1,
 1, -1, -1, -1, 1,
 1, -1, -1, -1, 1,
 -1, 1, 1, 1, -1 },

 {1, 1, 1, 1, 1,
 -1, -1, -1, -1, 1,
 -1, -1, -1, 1, -1,
 -1, -1, 1, -1, -1,
```

```
-1, -1, 1, -1, -1,
-1, -1, 1, -1, -1,
-1, -1, 1, -1, -1 },

{-1, 1, 1, 1, -1,
1, -1, -1, -1, 1,
1, -1, -1, -1, 1,
-1, 1, 1, 1, -1,
1, -1, -1, -1, 1,
1, -1, -1, -1, 1,
-1, 1, 1, 1, -1 },

{-1, 1, 1, 1, -1,
1, -1, -1, -1, 1,
1, -1, -1, -1, 1,
-1, 1, 1, 1, 1,
-1, -1, -1, -1, 1,
-1, -1, -1, -1, 1,
1, 1, 1, 1, -1 },

{-1, -1, 1, -1, -1, //A
-1, 1, -1, 1, -1,
1, -1, -1, -1, 1,
1, -1, -1, -1, 1,
1, 1, 1, 1, 1,
1, -1, -1, -1, 1,
```

```
                        1, -1, -1, -1, 1 }};


int YDesejado[11][11] = {{1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{-1, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{-1, -1, 1, -1, -1, -1, -1, -1, -1, -1, -1},
{-1, -1, -1, 1, -1, -1, -1, -1, -1, -1, -1},
{-1, -1, -1, -1, 1, -1, -1, -1, -1, -1, -1},
{-1, -1, -1, -1, -1, 1, -1, -1, -1, -1, -1},
{-1, -1, -1, -1, -1, -1, 1, -1, -1, -1, -1},
{-1, -1, -1, -1, -1, -1, -1, 1, -1, -1, -1},
{-1, -1, -1, -1, -1, -1, -1, -1, 1, -1, -1},
{-1, -1, -1, -1, -1, -1, -1, -1, -1, 1, -1},
{1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}};



f = fopen("SAIDA.txt", "w");

cudaMemcpy(_Peso, Peso, size, cudaMemcpyHostToDevice);

unsigned int timer = 0;
CUT_SAFE_CALL(cutCreateTimer(&timer));
CUT_SAFE_CALL(cutStartTimer(timer));

treinamento<<<1,11>>>(_Peso);
```

```
printf("\nTempo de computacao: %f (ms) \n\n", cutGetTimerValue(timer));
CUT_SAFE_CALL(cutDeleteTimer(timer));

cudaMemcpy(Peso, _Peso, size, cudaMemcpyDeviceToHost);



for (n=0; n<11; n++) {
yperceptron(ERRO, Peso, Ativa, Xin, YSaida, XQuantidade,
 YinQuantidade, Entrada[n], YDesejado[n], n);
}


fclose(f);

}
```